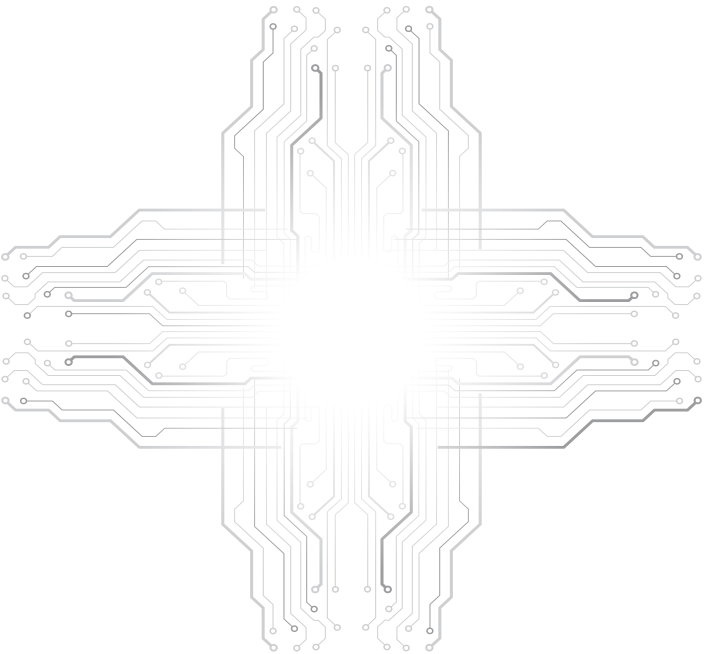


从芯片到云端
Python物联网全栈开发实践

刘凯 / 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

物联网开发重新定义了“全栈开发”的范围。Python 作为一门快速发展的语言，已经成为系统集成领域的优选语言之一，其可覆盖从电路逻辑设计到大数据分析的物联网端到端开发。各领域开发者可以利用 Python 交叉涉足物联网设备、边缘计算、云计算、数据分析的工程设计。

本书尝试让读者建立物联网设计的整体概念，从基础概念开始，到相关技术选型、开源工程、参考设计与经验分享。无论是物联网领域的创业者，还是系统架构师，都可从本书中获得灵感。本书对于嵌入式开发领域的开发者尤具学习价值，利用 Python 可加快开发迭代速度、降低开发成本，并可以基于嵌入式 Python 建立完整的物联网软硬件生态。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

从芯片到云端：Python 物联网全栈开发实践 / 刘凯著. —北京：电子工业出版社，2018.1
ISBN 978-7-121-31127-7

Ⅰ. ①从… Ⅱ. ①刘… Ⅲ. ①互联网络－应用－程序设计②智能技术－应用－程序设计
Ⅳ. ①TP393.409 ②TP18

中国版本图书馆 CIP 数据核字（2017）第 057641 号

策划编辑：张春雨
责任编辑：李云静
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：45.25　字数：1012 千字
版　　次：2018 年 1 月第 1 版
印　　次：2018 年 1 月第 1 次印刷
定　　价：119.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

推荐序

前几年国内引进了 Chris Anderson 的《创客：新工业革命》。打那时候开始，国内流行起“创客”风潮。“创客”这个词果真是一个洋气的舶来品，很多国人姑且把它看成硬件创业的预备役。但是大洋彼岸原产地的人们倒是朴实得可爱：织个毛衣，搞个室内大棚蔬菜。当然高科技类的自然少不了捣鼓一下机床，焊一块板子，这更像是一种 DIY 的怀旧文化：更加纯粹和快乐。做一名纯粹的创客并不容易，毕竟要抽出一定的时间和精力。直到现在我依然惦记着自己那台完成了一半的 3D 粉末打印机，而它就静静地躺在储物箱里。那时候的我已经开始为创业做前期准备，但商业项目和自己在创客空间玩的东西没啥关系，终究自娱自乐和商业有差别。

遇见 Allan 的时候，他也在努力从创客转变成创业者。我很惊诧于他虽然技术娴熟，也曾负责 NXP 产品技术与市场，却依然对技术保持着孩童般的初心，真的不多见啊。离开 NXP 后，Allan 决定成为一名自由职业者。靠着自己扎实的技术，从前端到后台，从硬件到软件，他一个人搞起了物联网的项目和产品。我们时不时在线上谈论可行的产品和市场策略。虽然我对硬件不熟悉，但是由于自己当时就职于 PTC，拥有些许物联网后台软件的认知，就这样我们相互参照着学习，并努力将其付诸实践。

2015 年 5 月，我离开了 PTC 并投身于机器视觉领域的创业，但依然保有对物联网的热情，尤其关注工业物联。而 Allan 在这几年的实践中积累了全栈开发的经验。终于有一天，他觉得是时候将他独自一人的全栈开发经验记录下来，并传播给这个领域的开发者了。我自然非常支持他，但是独自写一本技术类的书，这是多么考验人呀。之后和 Allan 的交流变少了，我想象得出他独自在房间码格子的情形。半年后，这本书的初稿终于扎扎实实地完成了。

创业者和分析师们总爱重复地问一个问题：物联网的风口有没有来？我们很难精准地去预判某个时间节点，但假如物联网是一个不远不近的方向的话，我们当下唯一能够做的便是顺着产业的脉搏而跳动。类比一下 PC 和移动互联网，我们依然处于物联网大规模商业化的早期。但是最终我们会迎来万物互联。让我激动的是万物互联的基础架构成熟后，在各个行业以及各个利基市场将会涌现出各种“新物种”，推动着商业和产业进一步提高效率，进一步打破边界。而对于希望投身于这个行业的技术人员来说，应该尽量抛弃这些华丽的时髦术语，回归技术本身。这本书平实地记录了读者需要了解和掌握的基础知识；与此同时，它从单一语言全栈开发

的概念出发梳理了一个完整的流程，而全局观的梳理能够更好地帮助技术人员去理解技术的本质。

技术总是在飞速地发展，书本记载的技能需要不断地升级更新。但是我能感受到 Allan 更希望传达的创客精神。创客愿意从零开始建一栋楼，他们或许不能建成一座摩天大厦，但至少也会筑成一幢别具一格的小楼房。这种纯粹的乐趣只有从动手实践中才体会得出来。但人们的生活节奏总是很匆忙，有这么一本类似于“宝典”的书，可以加快看官您动手的速度和效率。但愿您能享受从零开始搭建一个物联网项目或者产品的过程。

张成

浸梦信息科技有限公司创始人

自序

笔者曾经长期服务于微电子行业，现在从事物联网相关项目设计和咨询服务。

1995 年毕业后，笔者加入了飞利浦半导体上海技术中心，任软件工程师。在此期间的主要工作是使用汇编语言为国内客户进行各类显像管（CRT）彩电的固件开发。当时的技术环境，8051 都已经非常普及了，而飞利浦半导体彩电和固定电话技术方案中的控制器却依旧采用老旧的 Intel 8048 内核 MCU。该内核架构有许多限制：比如超过 2KB 代码需要切换代码段，缺乏高级语言支持，等等。虽然架构古老，但这个业务却一直是当时飞利浦半导体的“现金牛”（即主要利润来源）。

笔者后转入产品市场部，在那里可以接触到许多炙手可热的产品线。其中，笔者负责的产品如下。

- 8051 控制器：配合中国合作伙伴，如（北航）中国单片机实验室、南京万利、南京伟福、广州周立功等单位，合作推广 LPC764/9XX 及后来的 ARM LPC 系列。
- 通信产品：8048 内核电话机 MCU、传呼机、DECT 和中国数字无绳电话芯片组。
- 智能卡产品：包括电话卡、CPU 卡，以及最著名的 Mifare/Hitag RFID 和车用防盗钥匙。
- DSP 产品：Trimedia VLIW（超长指令集）DSP，用于视频电话和媒体处理。
- CPLD：低功耗 CoolRunner CPLD，后转售给 Xilinx。
- PDA：基于 MIPS R3000 内核的 Windows CE PDA 方案。

后来，笔者又重新拾起软件开发的工作，主要负责基于 8051/MIPS 的 LCD/DTV 的客户化固件开发。

应该这么说，在飞利浦的从业经验使得笔者积累了嵌入式开发经验，开阔了产品线视野，并积累了多方面的技术兴趣和行业人脉。同时，在开发这些嵌入式产品的过程中，笔者开始采用各类脚本语言来做代码生成和其他开发工具。

2008 年，飞利浦半导体部独立成为 NXP 公司之后，笔者开始了自己的创业之路。到目前为止，笔者独立设计过以下产品和参考设计：

- 基于 Cypress PSoC 的 RFID/UART/GPRS/TPMS 模块（C）。
- 基于 SDIO 闪存卡的 NFC 接口（FPGA/CPLD）。

- Wi-Fi 强制门户及热点分享网站（PHP）。
- GPRS+GPS AVL 设备及网站（C/Java/PHP）。
- TI C2800 DSP ANC 主动噪声抑制系统（C/ASM）。
- 网络爬虫，用于抓取超市的 POP 海报分发（Python+PHP）。
- Android 翻译 APP（Java）。
- 电子货架标签系统，第一个从设备到 APP 的完整原型设计（C+Python 网关）。
- GAP 创客电子模块，基于 NXP/Freescale/ST 的 M0/M3 处理器，并提供 Bootloader 和 ISP 软件（C/C++/Python）。
- 工业门物联网（C/C++/Python）。
- 呼吸机物联网（Python/Golang）。
- 电梯物联网（Python）。
- EPD 电子模块（C++/Python）。
- RFID 分类钱包（国家实用新型专利，已授权）。
- GPS 资产定位系统（C++/Python）。
- 电信 CDMA 基站监控设备（C++/Python）。
- VoLTE 高清语音监控设备（C++/Python）。
- 分级基金及股票监控报警系统（Python）。

离开 NXP 之后，笔者的设计不再受限于原公司的技术平台所涉及的消费电子产品领域，而是扩大到了互联网与物联网领域。笔者的个人体验是，无论是设备端还是服务器端，都有许多技术可以深入学习。但是两者融合，技术复杂度却呈现几何级数上升。

不同领域有不同的优势语言。一般来说，CPLD/FPGA 使用 VHDL/Verilog，MCU/SoC 固件开发使用 C/C++，桌面开发使用 C#/VB 等，服务器开发使用 Java/PHP/JavaScript/Python/Golang，手机 APP 使用 Java/Objective-C。

所以，笔者在工程实践中，一直在使用汇编语言/C/C++开发嵌入式系统固件，并使用 Perl/Python 脚本做开发支持工具，同时采用 PHP/Java/Python 做设备云和 Web 应用。一个完整的物联网应用涵盖许多环节：从数字逻辑电路设计，到硬件设计、固件设计、网关软件设计、服务器软件和网页设计、APP 设计，甚至模具的 3D 设计。出于工作的需要，即使环节长，笔者也不得不像“万金油”一样，亲自参与全过程的设计工作。虽然无奈，但笔者的修炼结果是，比一般硬件团队略懂服务器开发，比一般服务器/APP 开发团队略懂硬件开发，而且大致了解了物联网的许多具体技术。

笔者的个人体会是，物联网环节太长了！无论是设计、编码还是调试，物联网的庞杂特性都非常明显。首先设计和编码时间就很长，尤其在系统联合调试时，需要使用多种开发工具（仿真器、目标硬件、仪表、服务器、Web 控制台）。在这个阶段，有时候需要多台计算机才能够完

成调试任务。

以超市货架管理项目为例，其涉及 WSN 协议规划、节点端和网关端设备的固件开发和协议实现、服务器设计、手机 APP、条形码和二维码扫描。此项目笔者整整开发了一年才交付给客户，而且调试起来还挺麻烦。

个人单枪匹马，精力有限，无法同时兼顾所有环节，因而开发的项目格局不会太大。物联网开发应该为**团队合作**，甚至多个团队之间进行合作。每个团队对于各自的环节负责，做到接口标准化。这样才能够复用已有的经验和模式，并充分发挥其边际效应。即便是团队合作，也需要将自己使用的工具数量降低到最少，至少需要寻找到覆盖面较广的工具来开发。这也是现在许多“全栈”开发的目的。

采用单一语言做全栈开发

全栈开发最初出现在互联网行业，指的是能够同时开发网页前端和服务器后端。这包括能够做全栈开发的技术和掌握这些技术的工程师。该行业最典型的全栈开发语言是 JavaScript。

在物联网行业中，全栈开发的含义被延伸了。笔者推荐以 Python 作为全栈开发语言。本书的全栈开发涉及 IC（集成电路）设计、设备端（电路和系统）、服务器（含网页）端，以及移动端和数据分析端。使用单一语言可以多方面降低成本：

- 学习周期短，降低人力成本。
- 交付时间短，降低开发成本。
- 人力资源供应充分，降低人均开发成本。
- 容易形成生态，构建开发者生态圈，实现众包。
- 代码复用性强，代码可重用，开源市场有不少现成方案，可降低总体开发成本。
- 设备可以虚拟化，物理设备可以通过同一代码模拟出来，以加快工程启动周期，降低开发成本额和减少开发者间的责任推诿。

综上，物联网开发涉及面庞杂，开发周期长，寻找一种覆盖面广的编程语言和方法对企业和开发团队有现实意义。

Python 用于全栈开发

在笔者眼里，承担全栈开发的语言可以是 Java，也可以是 JavaScript，还可以是 Python。由于互联网的发展，加之 JavaScript 在前端语言中的优势地位，使得它开始延伸到了服务器后端和设备端。而 Java 原本就在设备端和服务器端都很有优势。从发展历史上看，在嵌入式平台中最早出现的是 Java，最近才开始出现 JavaScript 和 Lua 等动态语言。这都是服务器端企图深入到嵌入式行业的努力。至于 Python，由于其胶水特性，虽然性能不占优势，但是开发速度快，

比较适合做全栈的原型开发。

之所以出现企业和手机开发者力推 Java 开发，前端开发者力推 Node.js 开发前后端技术，某些群体力推 Go 的现象，除了技术本身的因素，许多情况下也是其教育背景和从业经历所导致的，即所谓出身和基因所决定的。在此，个人经历决定了笔者选择 Python 作为自己的主力开发语言。

曾经看过一个关于如何在 Java/JavaScript/C#/Python/Golang 等几种语言中选择一种作为主力编程语言的漫画式流程图。其中有一个选择：如果你喜欢乐高，那么请选择 Python。仔细想想，Python 的确很像乐高：

- 接口一致性高。
- 粗颗粒，构建速度快，适合原型。
- 有标准构件，如各种标准积木和标准库。
- 具备大量的定制构件。乐高中存在定制的主题人物和机器人组件，而 Python 也有大量的 C 扩展库和第三方应用库。

在我国的港台地区术语中，将 Integrated Circuit 翻译为“积体电路”，即积木化的电路。而新出现的各类集成技术，如 SoC/SiP，即系统芯片和系统封装，也是通过在电路 IP 领域和封装领域的创新来实现更大规模的电路整合。换而言之，不同规模的电路都是搭积木搭出来的。所以，半导体行业应该会比较偏爱类似于积木的 Python 语言。

回顾自己的从业和工程经历，大概以下是笔者偏爱 Python 开发的原因：

- 在固件开发中，接触到使用脚本语言（gawk）来设计代码生成器简化开发。
- 电子工程经验，接触并了解了许多企业的设备联网需求。
- 互联网工程开发经验，接触到了互联网/物联网领域的诸多环节。
- 在网站和 APP 开发经验中，不得不使用多种编程语言用于软件开发，了解工程管理的痛点。

Python 作为一种胶水语言，可在物联网及嵌入式系统中承担大量任务，并可以部分替代 VHDL/C/C++/Java/PHP/JavaScript 等各类语言，或者与这些语言进行互相调用。但是让一位工程师抛弃原有技术栈换用其他语言是困难的。最初，笔者只是在工程实践中发现 Python 的“出镜率”相当高。在一些小场景中笔者尝试使用 Python 开发后，积累了一定的使用经验。后来为了加速开发，笔者开始在客户工程中大量使用 Python 进行原型验证和服务器端开发。最终，Python 成为笔者的主力开发语言。

说起来，笔者本人的经历与 Python 的胶水特性很类似。笔者不能算是 Python 高手，所有的开发都是仅仅读了最基本的演示代码后就立即着手进行工程开发。笔者甚至连相关基本入门书都没有看完整，就着手使用 Python 构建系统。缺乏耐心的代价就是不断重新造轮子，即所谓的“重构”。不过，在不断换“轮子”的过程中笔者充分体会到了 Python 的各种优点。

无论是面向过程（POP）、面向对象（OOP）、面向切面（AOP），还是更加抽象的函数式编程（FP），Python 都可以支持。编程思想只有在项目中才能被不断加深理解。这一点，Python 对笔者的帮助非常大。之前，虽然也写过 C++/Java 程序，但是实际上真正让笔者完成从面向过程到面向对象编程思维转换的语言恰恰是 Python。

相当多的 Python 代码，一开始编写的时候是采用各类函数的面向控制编程，脚本化的倾向很强。随着代码复杂度的增加，笔者不得不反复重构代码，并主动引入了 OOP 的编程方法。体会了 OOP 的好处后，促使笔者反过来在设备端设计中重构 C++代码。在物联网服务器端开发时，笔者接受了面向切面的概念。在编写本书的时候，笔者又学习了函数式编程。在以后的开发中，笔者会有意识地增加更加抽象的编程思想以简化日常的编程设计。

实际上，在从事物联网的后端设计时，许多朋友强力推荐笔者使用 Java 进行开发。因为 Java 在企业级应用中积累了许多可重用的设计，是企业级应用的首选语言。但是笔者接触并熟悉 Python 之后，坚持使用 Python 开发网关和服务器。笔者发现使用什么语言真的不那么重要，只要自己熟悉就好。况且 Python 还可以用 Jython 来对接 Java 重用 Java 资源。说起来 Python 和 Java 是两种极端：Python 可以在许多语言中实现，而许多语言利用 Java VM 来运行。

熟悉了 Python 后，笔者就在日常工程中坚持使用 Python：在端口扩展与仿真、代码和文档生成、Web/IoT 服务器及嵌入式平台 Python VM 中都可以用到。而且每次开发后，总能够保留一些 Python 工具提交给开源社群，或者以后自己用。这也是不断自我强化的过程：熟悉一种工具，就会不断地利用这种工具去解决问题。

Python 的缺点及应对措施

首先，许多开发者认为 Python 的运行速度较慢，尤其无法与 C/C++编译的原生代码相比。由于 VM 的设计架构不同，Python，尤其是 CPython 比 Java/Lua 还要慢。Python 作为一种开源的语言，即使有各种各样的问题，利用开源社群的力量也可以更容易地找到各种解决方案。现在使用 JIT 技术的 PyPy 加速已经非常成熟，在许多场合都可以应用。Cython 也是一种性能极高的扩展，可以实现与 Golang 类似的性能。配合 libuv 异步库，Python 的网络性能不输于任何一种编程语言。性能不是唯一的要素，Python 的强大在于：**生态的完整，开发速度快，运行速度也很快**。

其次，Python 语言和代码本质上是开源的，所以更加适合开源软件使用。如果要实现闭源的商业化软件，可以将 Python 源码编译成 pyc，或使用各类 C 和其他扩展帮助保护核心设计，其代价是损失了 Python 跨平台的特性（除非扩展中也采用了某种跨平台技术，比如 JVM）。

最后，Python 的 GIL 问题也很有名，对多线程设计不利。解决方法有很多：多进程、协程及其他 Python 实现（如 Jython、PyPy、Cython 等）均可以回避这个问题。

前言

本书讲述如何以 Python 为主要编程语言，实现“从芯片到云端”的物联网应用系统快速开发和系统扩展。通过阅读本书，读者可以充分体会 Python 作为一门全栈开发语言，是如何在物联网的设备端、应用端、服务器端和数据端环节中发挥作用的。

编写本书的初衷是为了让准备或者已经从事物联网开发的读者能够通过 Python 语言缩短相关学习和开发周期；同时与大家分享一些经验教训，希望能够让读者在具体开发中回避各种“坑”。这不仅对开发团队，对于企业甚至投资者决策也是有益的。

大多数物联网相关书籍比较关注物联网系统和服务器端设计，但是物联网与互联网的设计差别在于：物联网系统设计受限于有限的设备计算能力、巨大的连接数量、独特的数据特征。所以完整的物联网系统设计需要考虑的要素比互联网更多，需要掌握的知识面既广且深。如何在短期内实现系统上线，并安全、平滑地实现规模扩展，一直是大家思考的问题。开发者可以采用的对策如下：

- 减少开发语言和工具种类。
- 使用成熟的参考设计和编程框架。
- 使用主流的云计算服务和可扩展的系统设计。
- 开源硬件、软件设计和并行开发模式。

有许多事情“开弓没有回头箭”。物联网的最大特点是大量的定制需求，而且上下环节的衔接往往存在技术依赖性，某个环节的决定往往会对其他环节的实施带来很大的影响，并可能造成开发团队间的责任推诿。这需要系统设计者事前做许多调研功课。笔者专注于设备域和服务器域，但本书力求带来更宽的视野，包括物联网相关的应用、产品和生态，介绍不同的系统架构和云计算服务，并在不同的技术选项中推荐几种比较适合工程实施和实际需求的主流组合。

在收集资料的过程中，笔者发现 Python 作为一门通用编程语言，应用范围非常宽泛。相信本书内容中有许多物联网相关的 Python 应用是出乎大多数人意料的：

- 支持 SPICE/IBIS 仿真与 VHDL 设计和电路的自动测试。
- 可以在许多流行的 8/16/32 MCU 上运行，包括 AVR/PIC/ARM/MIPS。
- 支持绝大多数 MCU/MPU/CPU 的外设和工业总线，而且编程接口非常灵活。

- 可以在各种类型的 Linux 上运行多种 Python 运行环境，包括 CPython、Jython 和各类嵌入式 Python。
- 通过 Jython 运行于 Java Runtime 中，与 Java 类库完美结合，切入企业级应用和大数据分析。
- 可以跨平台开发桌面应用和手机应用。
- 大量现成的网络安全和分析工具，可帮助开发者定位通信报文错误，或寻找系统安全漏洞。
- 提供大量的辅助工具，包括文档、软件工程、虚拟仪器、媒体处理等，为此笔者特地预留了第 8 章进行罗列。
- 物联网网关、服务器架构、数据分析和可视化、虚拟设备、通信协议定制等领域开发效率超高。

从 SPICE/VHDL 开始，到服务器，Python 实现了“从芯片到云端”的全栈开发。笔者希望这些内容和案例能够帮助开发者在启动项目前对开发有全局性的了解，并做出正确选择。

同时，本书的写作过程采用了 Python 相关工具，也是“全栈开发”实例之一。

- 格式：将 Python 文档中常见的 reST/Markdown 作为基础书写格式。
- 编译：采用 Sphinx 将 reST 章节编译成流行的 HTML 网页、ePub 电子书籍。
- 转换：采用 Pandoc（Haskell）转换成交付给出版社的 docx 主流文档格式。

读者可以将本书看作单一编程语言的物联网应用小百科，通过书中的简单例子大致了解物联网的开发流程，并可以根据自己的兴趣，在每章的延伸阅读[1]清单中深入探索、掌握物联网开发技术的具体实现细节。

目标读者群

本书的目标读者群是以下两大类开发者。

- 互联网开发团队：熟悉移动端 APP 的开发、服务器架构和网页前端开发，但对于传统制造业的技术领域，如芯片设计、硬件设计、固件设计、硬件系统集成，以及批量生产和库存管理缺乏足够的了解。
- 设备开发团队：主要是传统制造业产业链中的半导体供应商、独立设计公司、设备制造商。他们熟悉硬件设备的设计和流程，但普遍对于互联网应用和物联网架构缺乏足够的了解。

当前的制造业变化趋势是，设计与平台标准化，导致产品同质化竞争严重。这使得传统制造业在市场中逐渐丧失了议价权和话语权，处于被整合的被动地位。这些企业和团队在物联网

1 因篇幅所限，各章“延伸阅读”部分放于 www.broadview.com.cn/31127，请自行下载。

时代异常焦虑，急需掌握数据接入和数据分析技术，以增加市场份额，并提升市场竞争力和议价权。本书第 9 章主要讲述物联网服务器后端开发，可以帮助传统制造业了解服务器端和数据端的发展趋势、大致的技术方案构成，并可以利用 Python 做些简单的设备测试。

除了工业物联网、行业物联网外，消费端智能硬件领域的物联网开发案例非常多，这是市场热点之一。许多创业团队虽然可以自行设计 APP，搭建服务器，但是团队往往缺乏设备端制造经验，并仍在各类硬件问题中艰苦跋涉，苦苦摸索。本书在第 4 章中介绍了成熟的元器件、连接模块和实时操作系统，配合 Python 快速原型开发能力，让项目可以快速上市之余，还可以为设备添加各种“智能”应用。

此外，许多读者可能希望从全局角度了解物联网应用、各类技术方案甄选标准，以及具体技术细节。本书也尽可能地进行罗列，并就一些常见问题特别加以说明。

总的来说，本书适合对物联网及相关热点，如智能硬件、工业 4.0、万物互联的应用与实现技术感兴趣的人群阅读。目标读者群除了互联网从业者、微电子和 OEM/ODM 制造商、应用系统集成商，还包括学生、教师、创客、极客、Python 语言爱好者、产品经理、项目经理，企业高管和创投基金经理等。

最低阅读要求

由于本书是一本技术书籍，因此需要读者具备一定的编程经验和技术热情。如果读者对于 Python 基本语法有一定的了解那就更棒了。即便没有 Python 的使用经验，相信 Python 易学易用的特点也可以让读者很快入门。

此外，由于代码中大部分采用英语注释，因此需要读者具备基本的英语阅读能力。

本书的目的

核心目的

- 为应用开发团队提供设备端硬件、固件开发流程和开发工具方面的工程建议，并提供一些可以用于与服务器对接的硬件平台和参考设计。
- 为设备开发团队，提供服务器前后端/移动端的系统架构、开发框架、生态平台方面的工程开发建议，提供可以不断升级的可扩展架构和开发路径，以满足产品从原型测试、中试、量产到分布式规模生产系统整个产品生命周期的需求。

其他目的

- 分享基于 IaaS/PaaS 云计算平台的服务器开发经验，包括设备云、应用云和大数据服务。
- 分享可快速部署的物联网网关（Gateway）、边缘服务器（Edge Server）原型设计。

- 汇集 Python 在计算机系统中方方面面的应用信息，并持续更新。
- 吸引各方合力推动 Python 在嵌入式虚拟机/网关/服务器/大数据分析方面的开源活动。

本书内容安排

物联网环节长、技术庞杂，涉及的每种技术领域都值得大家仔细钻研学习。可以这么说，许多话题和技术都可以单独出一本书。所以本书力求在有限的篇幅内，突出物联网特征并使用 Python 落地生根，使之成为快速开发迭代的基础；与此同时提示在 Python 应用中可能遇到的问题和解决方案，以降低读者的学习成本。

本书以数据的流动方向，即数据的设备端采集、服务器接入、转发、分析到用户端的呈现为主线，并以 Python 语言从入门到各个技术栈中的应用作为辅线来安排章节。

本书内容编排经过多次斟酌和修改，最终按照以下顺序介绍。

章　　节	简　　介
第 1 章　物联网简介	概述物联网的定义、发展趋势以及物联网应用与技术等
第 2 章　Python 语言基础	数据类型、数据结构、内置函数和标准库
第 3 章　Python 语言进阶	多种实现、与其他语言的接口、物联网常见技巧
第 4 章　嵌入式系统开发	数字逻辑与模拟电路设计、C/C++固件开发以及主流的平台与供应商
第 5 章　设备连接和编程接口	物联网的多种连接性与编程接口以及 Python 支持包
第 6 章　嵌入式 Python 虚拟机	深嵌入式、嵌入式 Linux 最小系统以及各类 Python 虚拟机实现、演示
第 7 章　Python 应用 APP	在主流桌面操作系统和移动端中的 Python APP 开发
第 8 章　Python 开发辅助支持	在物联网开发环节中的原型验证、虚拟设备、数据分析等多个方面的 Python 开发工具
第 9 章　物联网服务器端设计	物联网网关、边缘服务器、Web/IoT 服务、开发框架和连接选项
第 10 章　融合应用与数据分析	科学计算、数据统计、数据挖掘和大数据分析平台和工具，以及数据可视化

除了本书内容，笔者还整理了诸多书中提到的 Python 扩展包和演示代码，并计划依托出版社网站和其他互联网服务进行分发。本书为笔者一个人写就，缺少专家进行校对，本人水平有限，书中难免有疏漏、错误，欢迎读者指正。但笔者精力亦有限，无法一一回复，祈谅。

本书未包括的内容

因为篇幅的限制，也因为物联网的特性，所以本书安排的内容比较繁杂。本书未能针对特定硬件、软件、云服务展开，也没有针对物联网提供完整的开源设计。这些希望读者在书本之外展开。本书出版后会依托各类互联网服务（如 GitHub、社交网站和 BBS）展开后续的开源设计活动。

软硬件环境

除非特殊应用和声明，本书主要的操作环境为 Windows 7（64 位）及 Ubuntu Linux 12.04（32/64 位）。Python 版本为 V2.7.11 和 V3.5。

在微控制器方面笔者推荐 ARM mbed 兼容的 LPC/STM/KL 开发板，或 Arduino；对于卡片电脑，推荐树莓派或者兼容的国产 Linux SBC；对于 MicroPython，推荐在 STM32F4XX/ESP8266 开发板上运行。

版权声明

本书所附代码和硬件，凡是笔者所做，皆采用 LGPL 协议，读者可以自由用于任意目的；其余软件和硬件，请参考各自官网中的版权声明。本书引用的图片、代码、图表等，其版权皆归属于所属公司、网站和个人。本书引用这些资源主要用于说明目的，且尽量在每章延伸阅读中标明出处。如有遗漏，请联络笔者本人。

感谢

本书付梓需要感谢许多机构和个人。

- 知乎网站：本书的创作主题来源于笔者在知乎上的提问，并得到了知乎网友（包括出版社编辑）的热心解答、正面鼓励和推动才能够走到这一步。
- 张春雨先生（永恒的侠少），电子工业出版社的策划编辑：在知乎上遇见后，你一路推动本书的出版。感谢你的耐心和鼓励。
- 张成先生，物联网创业伙伴：你不断鼓励笔者继续深入物联网开发，并拨冗为本书作序。
- 李云静编辑在本书成书过程中对我这一新手作者给予了极大的耐心，反复校对、纠错，感谢你和团队的付出。
- EEWorld 编辑 nmg 和版主 dcexpert：你们提供了宝贵的 MicroPython pyboard 开发板。
- 诸多开源项目的作者们：感谢大家对于开源软硬件项目的热情和不厌其烦解答问题的耐心。

芯片供应商及分销商：

- NXP（恩智浦）——感谢免费提供 LPC 系列开发板。
- Freescale（飞思卡尔，已与 NXP 合并）——感谢慷慨提供大量 KL25 的样片及技术支持。
- TI（德州仪器）及分销商 Serial（新晔科技）——感谢提供 WSN 技术支持。
- Cypress——感谢免费提供 PSoC 开发板。
- Fujitsu（富士通）——感谢友情提供 FeRAM RFID。

- Atmel（爱特梅尔）——感谢免费提供 MCU、Wi-Fi 和 Crypto 产品开发板。
- Nuvoton（新唐科技）——感谢免费提供 MCU 开发板。

最后感谢自己的父母和妻子，忍受笔者在放弃其他工程开发的情况下编写本书。
轻吻女儿 Kirin，谢谢你的耐心等候。

刘凯（奕辰，Allan K Liu）
2017 年秋，上海

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书提供示例代码及资源文件，可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31127

目录

第 1 章　物联网简介 1

1.1　物联网定义 1
1.2　物联网发展趋势 1
1.3　物联网应用与技术 2
1.3.1　物联网核心价值 2
1.3.2　物联网发展阶段 3
1.3.3　物联网分层 5
1.3.4　物联网数据传输与网络拓扑 5
1.3.5　物联网实施所需技术栈 8
1.3.6　标准、现状与未来 10
1.4　本章小结 16

第 2 章　Python 语言基础 17

2.1　Python 的由来与特征 19
2.1.1　概述 19
2.1.2　设计定位与哲学 19
2.1.3　优点与缺点 20
2.2　Python 与物联网开发 22
2.3　获取 Python 资源 24
2.3.1　Python 主程序 24
2.3.2　Python 文档 24
2.3.3　Python PyPI 24
2.4　Python 解释器运行环境 26
2.4.1　REPL 交互模式 26

2.4.2 直接运行与模块运行 26
2.4.3 脚本文件直接运行 27
2.4.4 源程序文字编码与结束符 28
2.5 Python 类型与语法 29
2.5.1 动态类型 29
2.5.2 传值与传引用 30
2.5.3 数据类型 31
2.5.4 内置类型 32
2.5.5 内置类型的普适操作 34
2.5.6 数值类型 35
2.5.7 布尔类型 37
2.5.8 迭代器类型 37
2.5.9 生成器类型 38
2.5.10 yield 表达式 39
2.5.11 序列类型 39
2.5.12 set 集合类型 54
2.5.13 映射类型 55
2.5.14 其他类型 56
2.5.15 控制流 59
2.5.16 内置函数 61
2.5.17 用户自定义函数 62
2.5.18 模块 65
2.5.19 输入/输出 68
2.5.20 面向对象编程 74
2.5.21 进程和线程 82
2.5.22 错误和异常 90
2.6 Python 标准库概览 93
2.7 本章小结 94

第 3 章 Python 语言进阶 95

3.1 HOWTO：常见任务和解决方案 95
3.1.1 数据类型转换 96
3.1.2 数据的调试打印 100

3.1.3 数据类型资源优化 102
3.1.4 数据结构与算法 102
3.1.5 数据缓存 103
3.1.6 数据多路复用和解复用 104
3.1.7 数据序列化和反序列化 107
3.1.8 数据压缩和解压缩 119
3.1.9 数据加密 120
3.1.10 数据传输 121
3.1.11 数据后处理 121
3.1.12 数据持久化 121
3.1.13 数据交换 122
3.2 HOWTO：函数式编程 123
3.2.1 高阶函数 123
3.2.2 map 函数 124
3.2.3 reduce 函数 124
3.2.4 filter 函数 124
3.2.5 sorted 函数 125
3.2.6 返回函数 125
3.2.7 闭包 126
3.2.8 匿名函数 126
3.2.9 装饰器 127
3.3 HOWTO：并发运行模型 131
3.3.1 协程 131
3.3.2 I/O 模型 134
3.4 HOWTO：日期与时间 136
3.4.1 类型转换 136
3.4.2 时区的处理 138
3.5 Python 版本迁移 139
3.5.1 Python 2 与 Python 3 的区别 140
3.5.2 Python 2 到 Python 3 的流程 140
3.5.3 多个 Python 版本共存 140
3.5.4 virtualenv 141
3.5.5 Windows 多个版本共存 141

3.5.6 Linux 多个版本共存 142
3.6 其他常见技巧 143
3.6.1 常数类型的模拟 143
3.6.2 枚举类型的模拟 143
3.6.3 开发自定义模块 144
3.7 Python 与其他语言 145
3.8 Python 语言扩展 151
3.8.1 C 语言扩展 Python 151
3.8.2 ctypes 访问 Windows DLL 153
3.8.3 Jython 访问 Java 类 154
3.8.4 IronPython 访问.NET 155
3.9 Python 加速 157
3.9.1 PyPy 158
3.9.2 Cython 159
3.9.3 PyCUDA 159
3.9.4 PyOpenCL 159
3.9.5 Theano 159
3.9.6 Nuitka 159
3.10 本章小结 160

第 4 章 嵌入式系统开发 161

4.1 嵌入式系统硬件分类 162
4.1.1 MCU 162
4.1.2 MPU 163
4.1.3 DSP 163
4.1.4 SMP 164
4.1.5 异构大小核 164
4.1.6 FPGA 原型 165
4.1.7 SoPC 165
4.1.8 GPU 167
4.1.9 哈佛结构和冯·诺依曼结构 168
4.2 电路原型设计 168

4.2.1 集成电路设计流程 170
4.2.2 模拟电路原型设计 170
4.2.3 数字电路原型设计 175
4.3 常见嵌入式微控制器（MCU） 179
4.3.1 MCU 市场状况 179
4.3.2 Arduino/Wiring 180
4.3.3 ARM mbed 181
4.3.4 设计专属架构和专属 MCU 182
4.3.5 ARM MCU 差异化竞争 182
4.4 常见嵌入式处理器和主板 184
4.4.1 ARM 架构 185
4.4.2 其余的 ARM Linux 主板 188
4.4.3 MIPS 开发板 190
4.4.4 x86 mini-ITX 191
4.5 常见传感器和执行器 192
4.5.1 虚拟传感器 193
4.5.2 智能传感器 193
4.5.3 专用传感器 194
4.5.4 执行器 195
4.6 物联网通信集成电路 196
4.7 嵌入式系统开发语言演进 197
4.7.1 从汇编到嵌入式 C 197
4.7.2 从 C 到 C++ 199
4.7.3 压缩 C++的系统消耗 199
4.7.4 C++适合物联网开发 200
4.8 C/C++的编程模式和技巧 204
4.8.1 C/C++设计模式 205
4.8.2 回调函数 206
4.8.3 有限状态机模型 209
4.8.4 善用结构体 211
4.8.5 C/C++协程 214
4.9 开发生态选择 215

4.9.1 工业标准与厂家私有指令集架构 215
4.9.2 硬件与软件平台选择 215
4.9.3 编译器选择 216
4.10 常见操作系统 217
4.10.1 无操作系统 217
4.10.2 RTOS 的优势 218
4.10.3 uC/OS 219
4.10.4 Keil RTX 219
4.10.5 mbed RTOS 与 mbed OS 220
4.10.6 FreeRTOS 221
4.10.7 Linux 是开发复杂联网设备的现实选择 222
4.11 物联网中间件 227
4.11.1 WSN 堆栈 227
4.11.2 TCP/IP 227
4.11.3 USB 227
4.11.4 FAT/FS 228
4.11.5 GUI 228
4.11.6 Terminal 228
4.11.7 MQTT 228
4.11.8 CoAP 229
4.12 物联网安全性 230
4.12.1 安全相关芯片 230
4.12.2 安全中间件 231
4.12.3 Python 安全算法 232
4.13 设备固件更新 232
4.13.1 固件更新技术发展史 232
4.13.2 本地固件更新 234
4.13.3 远程固件更新 234
4.13.4 固件升级定制 234
4.14 各类串口实现联网 235
4.14.1 串口协议的选择 235
4.14.2 模拟串口设备 236

4.14.3 其他类型虚拟设备 238
4.14.4 ISP 编程器 238
4.14.5 串口设备监控器 239
4.15 本章小结 239

第 5 章 设备连接和编程接口 240

5.1 设备连接概述 240
5.1.1 嵌入式系统连接层次 240
5.1.2 选择正确的连接方案 241
5.1.3 具体落实连接设计 241
5.1.4 本章内容安排 242
5.2 连接能力汇总 242
5.2.1 连接由芯片开始 243
5.2.2 芯片内部系统总线 245
5.2.3 芯片间连接技术 246
5.2.4 设备间连接 249
5.2.5 设备组网 250
5.2.6 设备组网与联网的无线技术 253
5.2.7 连接性回顾 266
5.3 Linux 文件系统 266
5.3.1 设备即文件 266
5.3.2 设备文件系统 267
5.3.3 Linux 设备文件的演变 268
5.3.4 文件 I/O 操作 271
5.3.5 Linux 硬件编程 272
5.4 并行接口 273
5.4.1 老旧的 PC 并行接口 274
5.4.2 高速总线 274
5.4.3 GPIO 274
5.4.4 Linux 访问 GPIO 275
5.4.5 GPIO 的 Python 包 276
5.5 串行接口 277
5.5.1 异步通信串行口 277

5.5.2 I2C 总线 284
5.5.3 SPI 总线 290
5.5.4 与其他硬件平台相关的 Python 包 294
5.6 USB 总线 296
5.6.1 USB Endpoints 297
5.6.2 USB Device/Host/OTG 297
5.6.3 USB 3.0 297
5.6.4 libUSB 297
5.6.5 PyUSB 298
5.6.6 标准化 USB 桥接 299
5.6.7 与 USB 相关的其他设计 301
5.7 Linux 网络设备驱动 301
5.7.1 TCP/IP 套接字编程 301
5.7.2 IEEE 802.3 到 IEEE 802.11 302
5.7.3 网络通信实现方案 302
5.7.4 私有通信协议栈 305
5.7.5 短距离无线连接 307
5.8 工业总线 310
5.8.1 CAN 总线 310
5.8.2 LIN 总线 312
5.8.3 其他 ASIC 313
5.8.4 定制 Python 扩展 313
5.8.5 Windows DLL 314
5.9 本章小结 314

第 6 章 嵌入式 Python 虚拟机 315

6.1 嵌入式高级语言平台大荟萃 315
6.1.1 高级语言与二次开发 315
6.1.2 BASIC 319
6.1.3 Java 319
6.1.4 Lua 322
6.1.5 JavaScript 322
6.1.6 .NET 323

6.2 前一代 Python 虚拟机 323
6.2.1 Telit GPRS 模块 323
6.2.2 Symbian 325
6.2.3 Windows CE 325
6.2.4 OpenMoko 325
6.3 深嵌入式 Python 平台 326
6.3.1 LEGO EV3 326
6.3.2 TinyPy 326
6.3.3 嵌入式 Python 的局限 327
6.4 PyMite 328
6.4.1 硬件平台 328
6.4.2 维护者 329
6.4.3 pymbed 分支 329
6.4.4 开发现状 331
6.4.5 文档 332
6.4.6 源码树 333
6.4.7 使用流程 335
6.4.8 实践 336
6.4.9 工程小结 337
6.4.10 网络资源 338
6.5 VIPER/Zerynth 338
6.5.1 硬件平台 339
6.5.2 Zerynth Studio 340
6.5.3 与标准 Python 的区别 341
6.5.4 快速启动 342
6.5.5 坎坷的使用过程 342
6.5.6 Zerynth 目录结构 343
6.5.7 硬件相关库 344
6.5.8 其他特性 355
6.6 MicroPython 356
6.6.1 工程背景知识 356
6.6.2 在线评估网页 358

6.6.3 官方硬件平台分支 358
6.6.4 衍生项目 359
6.6.5 UNIX 版本 360
6.6.6 MicroPython 库 363
6.6.7 STM32HAL 分支 365
6.6.8 NUCLEO-F401RE 适配 367
6.6.9 pyboard 评估 372
6.6.10 异步处理和中断处理 389
6.6.11 中断处理的普遍问题 392
6.6.12 使用心得 395
6.6.13 商品化与知识产权 396
6.6.14 BBC microbit 396
6.7 Linux 与 Python 398
6.7.1 Linux 中 Python 的运行环境 398
6.7.2 交叉编译 CPython 401
6.7.3 交叉编译 MicroPython 402
6.7.4 Jython 运行环境 404
6.7.5 Android SL4A 406
6.8 本章小结 407

第 7 章 Python 应用 APP 408

7.1 基于字符的人机界面 409
7.1.1 命令行参数 409
7.1.2 字符终端开发 410
7.1.3 ncurses 411
7.2 桌面 GUI 开发 412
7.2.1 Tkinter 413
7.2.2 wxPython 414
7.2.3 Boa Constructor 415
7.2.4 wxGlade 416
7.2.5 PyGTK 417
7.2.6 PyQt 419
7.2.7 PySide 420

7.2.8 Enthought 421
7.2.9 Cocoa+PyObjC 423
7.2.10 Java AWT 424
7.2.11 IronPython 与 WPF 425
7.2.12 其他 UI 425
7.3 本地 Web GUI 426
7.3.1 与 WebKit 相关的 Python 包 427
7.3.2 OneRing 427
7.3.3 Pyjs 427
7.3.4 Python Flexx 428
7.4 本地可执行文件 429
7.4.1 Linux 可执行文件 429
7.4.2 Mac OS X 应用程序包 430
7.4.3 Windows 可执行文件 430
7.4.4 pyinstaller 430
7.4.5 py2exe 430
7.4.6 py2app 430
7.4.7 cx_Freeze 431
7.4.8 Windows 系统服务 431
7.4.9 Windows 定时任务 432
7.4.10 Linux 系统服务 433
7.4.11 Linux 定时任务 435
7.5 移动 APP 开发 436
7.5.1 响应式网页 437
7.5.2 PhoneGAP 应用开发 437
7.5.3 SL4A 437
7.5.4 QPython 开发 441
7.5.5 Kivy 443
7.5.6 其他开发方式 449
7.6 本章小结 449

第 8 章 Python 开发辅助支持 451

8.1 物联网开发需要不断优化 452

8.2 专属小工具 452
8.2.1 单位转化器 453
8.2.2 内码转换器 454
8.2.3 其他编码转换 455
8.3 原型验证 458
8.4 代码生成器 459
8.5 软件测试 461
8.5.1 unittest 单元测试 462
8.5.2 socket 压力测试 462
8.5.3 urllib2 远程记录 463
8.5.4 PCBA 测试 466
8.6 文档生成器 468
8.6.1 文档格式 469
8.6.2 文档生成工具 473
8.7 文档操纵 477
8.7.1 Doc 文档操纵 477
8.7.2 Excel 表格操纵 478
8.8 国际化与本地化 479
8.8.1 gettext 479
8.8.2 Web 多语种切换 482
8.8.3 字库文件生成器 482
8.8.4 GB2312 点阵字库提取 482
8.8.5 TTF 字库提取 483
8.9 配置管理 484
8.9.1 软件配置管理 484
8.9.2 软件配置管理自动化 485
8.9.3 Git Bash 485
8.9.4 Dulwich/Gittle 包 485
8.9.5 Python Subversion 包 486
8.9.6 watchdog 系统监控 486
8.10 数据与素材处理 486
8.10.1 二维码显示 486

8.10.2 多媒体相关软件包 490
8.10.3 地理位置 494
8.11 通信报文分析 495
8.11.1 PyShark 495
8.11.2 pypcapfile 497
8.11.3 scapy 和 scapy3k 497
8.11.4 pcap Web 分析 497
8.12 与 Arduino/mbed 相关的 Python 包 497
8.12.1 Arduino Prototyping 498
8.12.2 pyFirmata 501
8.12.3 Py2B 501
8.12.4 CmdMessager 501
8.12.5 mbed 504
8.12.6 mbed RPC 504
8.12.7 mbed-ls 505
8.12.8 Python-mbedtls 507
8.12.9 Python-xbee 508
8.13 虚拟仪器 509
8.13.1 实时显示波形 510
8.13.2 Instrumentino 510
8.13.3 Vipy 511
8.13.4 PyVISA 511
8.13.5 Pythics 512
8.14 3D/VR/AR 512
8.14.1 PyOpenGL 513
8.14.2 PySoy 514
8.14.3 VPython 514
8.14.4 Printrun 3D 打印 514
8.15 本章小结 515

第 9 章 物联网服务器端设计 516

9.1 物联网计算模型 517
9.1.1 云计算 517

9.1.2 Web PaaS 与 IoT PaaS 518
9.1.3 IoT PaaS 供应商 518
9.1.4 PaaS/IaaS 混合架构 524
9.1.5 雾计算 525
9.2 物联网与互联网设计异同 526
9.2.1 基础架构 526
9.2.2 标准化程度 527
9.2.3 业务模式 527
9.2.4 系统构成 527
9.2.5 设备接入协议 528
9.2.6 数据特性 529
9.2.7 系统架构 530
9.2.8 数据持久层 532
9.2.9 大数据分析架构 534
9.2.10 业务耦合与分离 534
9.2.11 业务与数据融合 535
9.2.12 认证授权与计费 535
9.3 物联网网关与边缘服务器 535
9.3.1 Python socket 服务器 536
9.3.2 pyserial RFC2217 536
9.3.3 SubGHz 网关 panStamp 537
9.3.4 Rascal micro 538
9.3.5 Java IoT 网关 539
9.4 物联网设备接入协议 540
9.4.1 异步通信框架 Twisted 541
9.4.2 Twisted 套接字服务器设计 544
9.4.3 物联网专用协议 558
9.4.4 CoAP 560
9.4.5 MQTT 564
9.4.6 mosquitto/paho 567
9.4.7 REST API 572
9.4.8 服务器数据推送技术 572
9.5 高可用性与高并发性 575

9.5.1 并行与并发计算 575
9.5.2 网络 I/O 模型分类 575
9.5.3 架构优化的路径 576
9.5.4 关系数据库系统 576
9.5.5 SQL/NoSQL/NewSQL 578
9.5.6 Redis 579
9.5.7 MongoDB 580
9.5.8 时序数据库 581
9.5.9 消息队列 583
9.6 业务与数据融合 585
9.6.1 网站权限管理 585
9.6.2 认证授权与计费 586
9.6.3 OpenID 587
9.6.4 OAUTH 587
9.6.5 OpenID 与 OAUTH 的异同 588
9.6.6 社交化硬件 588
9.7 Web 开发框架 589
9.7.1 MVC 模型 589
9.7.2 Web 开发流程 589
9.7.3 Python Web 百花齐放 590
9.7.4 Zope 591
9.7.5 Django 591
9.7.6 Flask 592
9.7.7 gevent 提升性能 593
9.7.8 异步 Web 框架 Tornado 593
9.7.9 异步网络框架 Twisted 593
9.7.10 异步 Web 框架 Cyclone 594
9.7.11 静态网页 594
9.7.12 TLS 安全网页 594
9.8 物联网安全 597
9.8.1 物联网安全现状堪忧 598
9.8.2 操作系统安全 598
9.8.3 数据缓存与数据持久层安全 599

9.8.4 Web 框架与容器安全 599
9.8.5 远程加载风险 600
9.8.6 Web 前端安全 600
9.8.7 传输层安全 601
9.9 服务器交付 603
9.9.1 虚拟机交付 603
9.9.2 Docker 容器交付 603
9.9.3 VirtualEnv 交付 605
9.10 服务器运维 605
9.10.1 Linux 定时任务 606
9.10.2 常见的定时任务 610
9.10.3 系统监控 611
9.10.4 集成化运维软件 613
9.11 物联网系统设计实践 614
9.11.1 服务器端需求分析 614
9.11.2 确定设备接入方式 616
9.11.3 物联网的实时要求 617
9.11.4 EPIC IoT 设备服务器 617
9.11.5 EPIC 架构优化 619
9.12 本章小结 625

第 10 章 融合应用与数据分析 626

10.1 物联网是可编程的 626
10.1.1 Web API 的“满汉全席” 627
10.1.2 Web API 技术演进 628
10.1.3 IoT Web API 的必要性 628
10.1.4 Device as a Service 629
10.2 数据统计、分析和挖掘 630
10.2.1 名词解释 630
10.2.2 术语小结 631
10.2.3 大数据分析 632
10.3 采集整理自有数据 633

10.3.1 原始设备数据 633
10.3.2 数据埋点 633
10.3.3 服务器端数据 634
10.3.4 需求确定分析方法 637
10.4 采集第三方数据 637
10.4.1 结构化数据 638
10.4.2 半结构化数据 638
10.4.3 非结构化数据 639
10.4.4 数据录入 644
10.4.5 数据融合 644
10.4.6 数据规整 646
10.4.7 数据交易 646
10.5 数据分析 647
10.5.1 常见编程语言 647
10.5.2 数据分析分类 647
10.5.3 科学计算数据分析工具 651
10.5.4 统计学数据分析工具 658
10.5.5 金融数据分析工具 659
10.5.6 大数据平台与生态 661
10.6 数据可视化 663
10.6.1 数据可视化的发展趋势 664
10.6.2 matplotlib 665
10.6.3 seaborn 665
10.6.4 mpld3 666
10.6.5 Chaco 667
10.6.6 Pygal 668
10.6.7 Plotly 670
10.6.8 TVTK 671
10.6.9 VPython 672
10.6.10 Folium 673
10.6.11 NetworkX 674
10.6.12 Bokeh 675
10.6.13 Mayavi 677

10.6.14 Vispy 679
10.6.15 MoviePy 680
10.6.16 其他新技术 681
10.7 本章小结 682

推荐书目与结束语 683

第 1 章

物联网简介

本章主要介绍物联网的定义及其发展趋势，以及物联网应用与技术等内容，以便读者对物联网有个大概了解。

1.1 物联网定义

到目前为止，已经有许多标准化组织、国家机构、大学、跨国公司、技术联盟和百科全书等都对“物联网”做了定义。总的来说，以描述性定义为多。

物联网（Internet of Things）是指通过各种网络连接技术和信息传感技术，将物理实体进行标识、分类、组网、联网，实现物与物、物与人之间的连接与互动，在数据采集基础上通过处理、分析、模式识别等智能算法，实现自动化管理、远程控制的计算机应用系统。

早期物联网定义多与 RFID 有关，用于实现物体和用户识别。现在加入了各类传感器、网络连接技术和智能算法，物联网的概念被大大拓宽了。

物联网的关键词是物和联网技术，其整合了许多 RFID、传感网、工业控制以及新型的云计算和人工智能技术。物联网成为 TMT（Telecommunication，Media，Technology，即电信、媒体、科技）相关行业对其他传统行业进行融合性技术升级的技术趋势。

对于金融市场来说，物联网是一种投资概念。但物联网不是一个单独的行业。目前被券商贴上“物联网”标签的企业，都是在日常生产或营销方式中采用了某种物联网技术的企业。这些行业可以分属于许多行业，如医疗、工业、物流，而不一定是 TMT 行业。

1.2 物联网发展趋势

和物联网有关的热门词汇有很多，比如智能硬件、工业 4.0、万物互联、互联网+等。可以预见会有更多时髦词汇不断出现，如果将这些概念视为果实和目标，物联网则是这些概念的技

术内核和基础。

随着市场对于物联网的持续关注和资金投入，物联网应用呈现出以下发展趋势。

- 碎片化：各种物联网应用百花齐放，而且技术选项越来越复杂。
- 标准化：芯片、连接技术、操作系统、设备平台、云计算平台、云计算服务呈现出标准化和同质化的趋势，而且迭代速度越来越快。
- 拟人化：物联网不再仅仅局限于设备组网、联网，配合云计算和边缘计算，系统内部和系统间的互动越来越像一种“生命体”。

所谓“拟人化”发展趋势，指物联网系统从功能上看越来越像一个生命体。虽然计算机系统与生物学中人的概念无法一一对应，但是这种发展趋势非常明显：

- 集成电路、微机电是物联网的终端设备物质基础，如同“蛋白质”。
- 创新电源设计为设备长期运行奠定了能源基础，如同“碳水化合物”。
- 传感器、执行设备用于感知环境和执行指令，如同“耳目”与“手足”。
- 组网、联网技术让离散设备组合成网络，如同“神经”。
- 云计算平台作为物联网控制中心，如同“大脑”。
- 云存储、大数据、机器学习、人工智能构成一个完整的应用，如同“大脑”的“记忆、观察、经验和判断”。
- 计算机系统病毒、设计缺陷，可以被视为“病毒”，“病菌”和“敏感源”。
- 不同应用通过 Web 服务彼此相连，构成了应用之间的复杂关系，此类关系可以是隶属、对等甚至敌对关系。
- 不断被标准化、虚拟化、抽象化的系统架构、硬件、数据、服务，使得应用可以被不断地复制和传递。

依托人类的群体智慧，许多无法计量的无机生命体正在形成，并且彼此相连，构成更加复杂的系统。读者也可以将本书视为物联网系统的 DNA，因为你们可能会将本书中某些设计应用在自己的物联网设计中。

1.3 物联网应用与技术

本节主要介绍物联网的应用价值、发展路径和常见网络拓扑。

1.3.1 物联网核心价值

曾经有个物联网的笑话：“我们为什么要联网？难道是离开家后想起自己忘记冲马桶，所以发个短信/微信来冲马桶？”这个冷笑话说明了消费品领域物联网的投资泡沫，尤其是智能硬件，

已经引发了大众对于物联网核心价值的误解。

采用物联网技术可以节省大量的人力资源、基础生产要素和环境使用成本。这是物联网的核心价值所在。但这仅仅是从零到一的一步，物联网的附加价值还需要若干年才可以显现。物联网在智慧城市、智慧交通、智能建筑、工业生产、农业生产、国防军工、商业分销、产品安全、基础建设、环保节能、教育培训、运动健身、财经金融、旅游酒店、产品防伪、医疗卫生等多方面的成功应用，从降低成本、提升附加值、提高良率、提高满意度、降低获客成本、提高管理水平等多个方面对系统的利益相关方提供了新的工具和解决方案。

一个物联网项目必然需要解决某个应用场景中利益相关方的痛点，才能够获得成功和普及。物联网是某种行业转型升级中的必要工具和抓手，而非终极目的。

1.3.2　物联网发展阶段

美国参数科技（PTC）公司因其旗下旗舰产品 Pro/E 而为人所知。作为一家长期服务于制造业产品生命周期以及计算机辅助设计领域的企业，PTC 还并购了物联网领军企业 ThingWorx 和 Axeda。PTC 的 CEO James Heppelmann 和哈佛大学教授 Michael Porter 合写了一篇文章：*How Smart, Connected Products Are Transforming Competition*（智能互联产品如何改变竞争方式）。该文刊登在 2014 年 11 月的《哈佛商业评论》上。文中揭示了物联网的产品发展阶段和企业的竞争战略，值得读者去仔细阅读。该篇文章还有姊妹篇：*How Smart, Connected Products Are Transforming Companies*（智能互联产品如何改变公司），刊登于 2015 年 9 月的《哈佛商业评论》上。

本节以图 1-1 所示的车联网产品与系统演变为例说明物联网对于产品的影响：

- 独立产品，如电气机械组合而成的汽车。
- 智能产品，如增设传感器和总线后，具备自我诊断能力，具备一定智能算法的汽车。
- 智能互联产品，具备远程连接、远程控制、无线定位的汽车和服务。
- 产品系统（system），通过 V2X 包括 V2V/V2R 通信方式，可以发展成为具备综合智能的车联网系统。
- 产品生态体系（system of systems），车联网系统可以接入 ITS 和其他业务系统如私车分享、车位分享、智能物流、车辆维修、保养等许多系统而成为更加庞大的车联网生态系统。

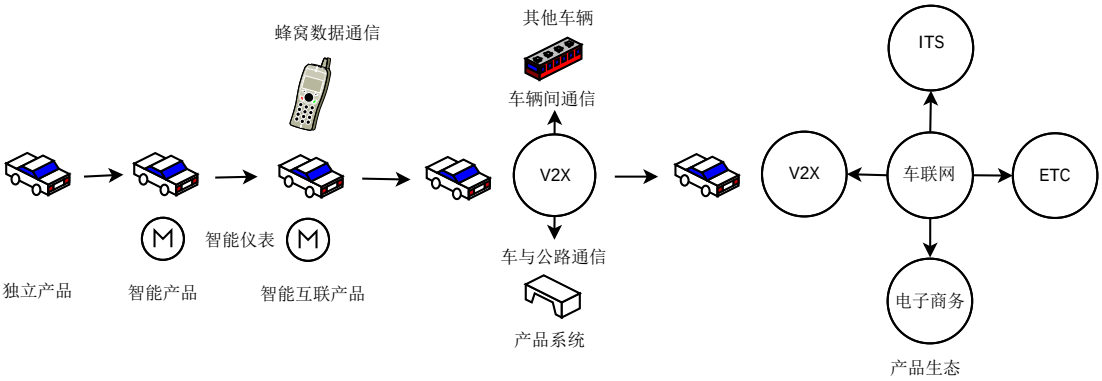


图 1-1　车联网产品与系统演变

2014 年是所谓的物联网（或称万物互联）元年，笔者也特地参加了在杭州互联网小镇的“万物互联大会”。笔者不同意物联网元年的说法。这抹杀了之前二十余年许多行业在物联网方面的努力。至少笔者手里最初用于 IP 网络的 MCU 硬件和固件是 1996 年的。如果将 AMD/Intel 的 x86 工控板、68K/SH/ARM/MIPS 的 PDA 和手掌游戏机联网也计算在内的话，物联网和互联网的发生、发展几乎是同步的。

物联网需要多种设备和技术的组合。笔者认为物联网是早就开始的分散性市场，目前在各个区域和各个细分市场存在许多小龙头企业。日后这些细分市场的领军企业如何和国内外的互联网巨型企业进行合作依然存在很大的变数。

不同的行业和应用场景，其物联网的发展阶段是参差不齐的。许多产品，如门锁，大多处于独立产品阶段。但是已经有企业开始推广智能锁了，无论是单机版还是联网版都有。如果有平台企业如房地产短租和车位转租，涉及移动互联网与资金交易，那么这个细分行业就会快速迈入产品系统和生态体系阶段。可以很清楚地看出来，一旦有领先的制造商开始提供智能化和智能互联产品，就会有平台供应商通过资本的力量快速接入该产品进行应用场景整合。

另外举个例子，大多数呼吸机（即鼾症治疗仪）没有联网，甚至该产品都没有普及。但已有客户委托笔者开发云平台以配合其智能互联呼吸机的销售。虽然供应商有自己的设备云平台，但是也必须和一些智能医疗的服务平台供应商合作。这个实例进一步说明，虽然平台供应商贴近最终市场，但是供应商必须拥有自己的智能互联产品和云平台构成产品系统，否则连和平台企业合作的资格都没有。即使设备供应商无法整合市场，也依然必须构建自己的物联网产品系统。

这两个例子说明：设备供应商做的是为现有设备提供智能和 IP 连接性，即智能互联产品阶段。而平台供应商提供的是针对应用场景的产品系统整合。一些电商企业则在建立基于某些消费场景的生态系统，比如智能家居、智能医疗，就是想要构建产品生态。

对于制造企业，千万不要止步于提供连接性；否则，会被沦为代工企业，丧失话语权和市场份额。当前最大的痛点还在于，设备制造商还无法专业高效地对现有设备进行升级改造，并将设备抽象成一种数据服务提供给自己的合作伙伴和用户。这也是笔者编写本书的核心目的之一。

1.3.3　物联网分层

物联网可以笼统地分为感知层、网络层和应用层。这种分类过于简单，在许多场景中需要很多附加说明：

- 感知层不仅仅有传感器，还必须有执行器。此外，还遗漏了雾计算模式的分布式智能，雾计算模式中还包含了部分应用层的计算任务。
- 感知层内的传输，如大量的 WSN 网络、现场控制总线网络，应该归于网络层还是独立定义感知网络层，还无法定论。
- 应用层不仅仅是单一应用，还应该增加应用间的数据交换和交互。

另外，上述三层物联网模型还忽视了应用层之上的应用融合、行业融合、数据分析与挖掘、机器智能等，视野不够开阔。

1.3.4　物联网数据传输与网络拓扑

物联网与互联网不同，许多制造业都是重资产企业。2016 年最热的共享单车虽然也有一些企业参与，但是与千团大战相比就是小巫见大巫了。毕竟硬件设备试错成本太高了。所以，制造业等生产型企业的物联网确实可以长期维持去中心化和碎片化的特点，因为这些行业的感应层连接技术实在太分散，替换的成本太高。但是随着互联网的渗透，在行业内部出现了标准化、IP 化、消费化的趋势。

从目前来看，不同应用场景的物联网数据传输特性与网络拓扑存在着很大的关联性。

在如图 1-2 所示的示意图中，从数据的流转环节上对单一产品物联网系统的拓扑和系统构成进行了简单划分，我们可以从此图了解物联网的开发环节。

- Edge Node/Sensor Node，所谓边缘节点和传感器节点，往往就是各种独立的设备和工作单元。
- BAN/PAN/LAN/HAN，是 Body/Personal/Local/Home Area Network 的缩写，是各种近场通信的集合，包括有线和无线通信。
- Gateway，网关，指的是负责近场通信协议和服务器端之间的转换设备。
- WAN，广域网指的是各类连接到公众互联网的连接技术，比如蜂窝数据传输、FTTB 甚至卫星等。

- Cloud，就是云服务器，包括设备云、应用云、数据库等各类服务器。
- Application Action，指的是各类桌面和移动应用程序、第三方应用，以及规则引擎。
- Data Analytics（数据分析），大数据分析平台。在上述划分中，大数据是单独划分的数据端平台。

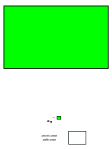


图 1-2　物联网数据传输与拓扑示意图

通过以上的网络拓扑，可以了解到，在边缘节点和传感器节点，可以根据网络覆盖特性采用点对点、星型、环型、总线型、分布式、树型或网状拓扑，但是其最终都将通过网关连接到服务器。单一物联网产品的网络拓扑从系统角度看是树型拓扑，其按照分级原则逐层上传归集数据。

图 1-2 中并非绝对的划分，许多离散连接设备比如共享单车可以跳过网关直接通过 WAN 连接到云。而且 BAN/PAN/LAN/HAN 与 WAN 部分有重叠的部分，比如 LoRaWAN 的通信堆栈与相关技术和许多 WSN 类似，但是由于距离长，在现在的划分中，将其归类在 LPWAN（Low Power WAN）中。现在我们可以看看在具体应用环境中网络传输拓扑的对应情况。

1.3.4.1　智能家居

智能家居以 Wi-Fi 为主要近场传输方式，BLE 次之，并形成以手机、路由器、智能电视，甚至温控器和无线音箱等设备为网关的应用模式。Z-Wave、Zigbee、SubGHz 和红外虽然在智能家居中有一定应用，但位于被边缘化的地位。

Bluetooth 5.0 之后增加的 Mesh 和 IPv6 对于日后的智能家居发展影响很大。在 Bluetooth 4.0 普及后，5.0 的升级换代还需要一些关键厂家如 Google 和 Apple 的支持，才能够走上普及之路。

1.3.4.2　智能农业

智能农业以 Wi-Fi 为主干，利用 SubGHz/2.4GHz 短距无线电以及 RS485 有线网络为分支，形成生产区域的大规模覆盖。由于经常处于蜂窝通信无法覆盖的区域，因此更远距离会采用 LoRa 等 LPWAN 形成连接。由于农业还需要细分为农林牧渔，而且不同农业地区的供电情况不同，因此往往采用定做的网关路由器将农业传感器和控制器连入 IP 网络。

1.3.4.3　智能工业

工业应用环境比农业环境更加恶劣多变。由于自动化领域的条块分割和硬实时安全要求，造成各个子行业中的各类设备采用的网络无法互联互通。传感器网络就存在不少种类：RS485/CAN/ProfitNet/ EtherCAT/ModBus 工业总线形成第一层传感器链接，工业的 WSN 也包括 Wireless HART/ISA100/Wireless-MBus/DASH7。此外，最近工业界也开始引入各种主流标准，比如以 6LowPAN/Zigbee 等作为新的设备连接方式。与农业方面一样，工业领域急切需要一个通用网关将林林总总的工业控制总线连入标准 IP 网络。

总的来说，工业、农业需要网关设备将私有网络转换成 IP 网络；而消费领域由于手机、平板电脑和智能电视、机顶盒等本身都已经 IP 化，同时具备 UI，因而有其特殊性：其既是终端，又可以是网关或路由器。从拓扑来看，这可以比工农业少一个层级。

为了便于操作人员利用手机对设备进行远程控制和访问，消费电子与工业、农业和专业领域的某些连接技术出现了彼此融合升级的趋势。消费市场领域针对 Mesh 推出了不少技术：BLE 推出 Mesh 和 IP 网络，6LowPAN 可以工作在 SubGHz 和 2.4GHz，等等。这些技术推出后对于市场的影响尚有待观察。

1.3.4.4　新型网络拓扑

前面所描述的网络层次和网络拓扑是针对传统网络而言的。随着物联网的发展，设备和数据整合将发生在各个节点。物联网发展的最大挑战不是简单地建立一个去中心化的物联网，而是建立一个规模可以不断拓展的通用物联网，同时保证隐私、安全和无须信任交易。

区块链（block chain），能提供一种无须信任单个节点，还能创建共识网络的方法。美国计算机科学家莱斯特 • 兰伯特在分布式计算中提出的“拜占庭容错系统”中指出，一群将军互不信任，其中一定有叛徒，但只要“怀有二心”的将军人数不大于将军总数的三分之一，计算机就有一个算法，保证将军们达成的共识是真实的。放置到物联网或互联网中，意味着即便网络中有不安全的节点，也可以保证信息的安全传送。

比特币使用算法工程保证整个网络的安全，设备能在金融市场中完全独立于任何人工干预。其底层采用了区块链算法。所以，比特币才能够成为没有任何央行发行的独立电子货币。

采用区块链算法，可以在一定程度上解决网络的身份认证问题。未来网络的拓扑将不再分为层次，而是网状拓扑（Mesh）。即便物联网遭到攻击，也会很快地恢复过来。这一点和现在以服务器为中心的星型拓扑是不一样的。好在网络升级主要改变的是传输协议和算法，而非硬件架构。

感兴趣的读者，可以去了解比特币、区块链和物联网的应用。

1.3.5 物联网实施所需技术栈

物联网的实施难度在于系统集成链条太长，工作量巨大。物联网各个环节的技术难度对于熟悉该环节的工程人员来说不算太高，所以大家往往误以为集成起来也会很容易。其实，做过系统集成就知道了——很累。

狭义的物联网以 RFID 为基础，而现在物联网已经成为一大类工程科技的大荟萃。解决设备联网的连接性、安全性后，大家期待的是商业模式和数据挖掘所带来的利润。虽然物联网开发的每个环节所需技术门槛并不高，但需要跨领域开发，加上万物互联的多态性、物联网拓扑的复杂性，再加上更为复杂的商业因素，使得物联网的专业性强、开发环节多、组合变化多、开发周期长、多种标准竞争激烈而诡谲多变。这使得物联网成为传统企业、中小企业无法短期切入的领域。

从实施角度来看，物联网需要开发者具备以下技术手段：

- 集成电路和数字模拟电路原型设计，采用 VHDL/Verilog 进行 FPGA/CPLD 设计。
- 嵌入式开发设计，含设备硬件和固件开发，使用 EDA 工具、C/C++编译器、JTAG/SWD 调试器，需要熟悉 TCP/IP 以及文件及操作系统底层。
- 软件开发，主要指桌面系统和手机 APP 开发，使用 Java/C#/Objective-C/ C++等。
- 服务器端开发，主要指 Web 服务器、IoT 网关，以及 Web 前端开发，使用 JavaScript/PHP 等多种语言）。
- 架构设计，主要指的是 WSN 网络的 MAC/LLC 等协议栈规划等，往往都是顶层设计后体现在固件和 IC（集成电路）设计中。

从事 IC 设计和架构设计的人员较少，大多数工程师从事的是嵌入式开发（含硬件和固件）、桌面软件开发和服务器端开发，以及 APP 开发。虽然开发手段很多，需要使用的编程语言也很多，但是 Python 作为一门胶水语言，可以像“万金油”一样在每个环节中使用。而且在大多数情况下，其开发效率更高，实施时间更短。

1.3.5.1 设备端

嵌入式开发属于电子工程和计算机科学的交叉学科，但是大部分相关从业人员的教育背景以电子工程专业为主。该领域和 IC 设计有部分交叉，FPGA 既可以使用 HDL 进行描述，也可以使用原理图工具输入，所以有软件和硬件背景的人均可担任。

嵌入式软件大致分为两类：

- MCU 开发，往往采用片内 RAM/ROM 存储器，采用 RTOS 或者 Round Robin 方式作为任务调度常见方式。目前，ARM 逐渐主导了 MCU 市场，许多原厂 IP 如 PIC/AVR/XA/6805/68K 逐渐退出市场。
- MPU 开发，往往采用片外 DRAM/Flash 存储器，主要采用较为复杂的操作系统，包括

Linux、Android 或者 VxWorks 等高级 RTOS。目前，ARM 基本上主导了该市场，但是 x86 和 MIPS 依然占据相当的市场份额。

嵌入式开发用 C/C++甚至汇编语言编写程序，比较麻烦的事情是调试困难，资源匮乏，缺乏操作系统支持，需要规划很多系统的细节，并要在不同的 API 和硬件细节间做出选择。

在嵌入式软件领域，使用并推广 Python 编程经验，是笔者最初的写作动力。笔者发现 Python 可以在 MCU/MPU 的嵌入式虚拟机、用户代码定制、开发支持、HDL 与数字逻辑设计、烧录工具、测试工具、生产工具、代码和文档生成、设备驱动、设备模拟及服务器仿真等各个方面加速整体开发进度，继而为从事嵌入式系统开发的同仁们提供帮助。

各类嵌入式 Python 虚拟机，如 PyMite/MicroPython/Zerynth 等，可以在 Cortex MCU 上直接运行，这是普通设备升级为智能设备的重要途径。开发者甚至消费者可以在这些设备中直接下载脚本运行。

同时，采用 RTOS+Python 或者 Linux+Python 的组合，也便于服务器/互联网行业的人士了解上游商业的生态，并构建设备联网快速原型作为参考设计提供给合作伙伴。

1.3.5.2　服务器端

嵌入式和服务器之间有一个比较大的分水岭，电子工程和计算机科学背景的人才往往无精力继续向对方领域发展。但两者间的桥梁是 Linux，了解 Linux 的工程师容易实现跨界开发。本书基于 Linux，提供了一些简单的服务器设计以及通用物联网网关设计，以满足各行业对于所谓交钥匙工程（Turnkey Solution）的巨大需求。

- 为了满足不同行业对于演示、小批量、中等批量的 OEM 需求，我们设计了一个可扩展的服务器架构 EPIC。
- EPIC 最小配置可以运行在各类手机、平板、高清机顶盒、MPU 开发板等嵌入式系统中，可以作为边缘服务器负责近场的区域服务。
- 中等规模可以实现单台单核服务器满足 1~20 000 用户/设备长连接。
- 更大的规模可以通过购买其他负载均衡、内存数据库、大数据数据库、分布式数据库等云计算组件实现百万级连接。
- 实现基础的数据统计、分析和可视化支持。
- 实现基础的 REST API，便于系统整合。
- 通过 Linux + Python 的组合，以及即插即用的 USB 模块，实现通用型网关的架构设计。

在服务器领域，存在着大量的编程工具和语言：Java/PHP/Golang/Ruby/JavaScript 等，但 Python 在服务器和服务器端编程语言中扮演着重要的角色。在开源服务器应用中存在着大量的 Python 服务器框架，其中包括 Django 和 Twisted，以及 Tornado 和 Zope。

1.3.5.3 客户端

现在的手机处理能力直逼桌面处理器，许多处理器如 Intel ATOM，之前是为上网本等桌面计算机系统开发的处理器，现在用于手机和平板电脑。所以桌面和手机客户端的开发资源上趋同，仅操作系统和开发环境有所区别。

这也是 Python 在手机中可以扮演多重角色的计算基础。Python 软件在手机中，既可以开发客户端软件（如 Kivy），又可以开发服务器软件（Twisted），也可以扮演网关角色（如标准套接字（socket）+ Java USB host API）。

在手机开发中，Nokia（诺基亚）的 Symbian 中曾经出现过 Python 的影子。OpenMOKO 也是一个例子。Android 中也出现了含 Python 在内的脚本引擎。但是总体而言，Python 在 Android/iOS 中的应用不算很普及。

在桌面应用程序中，使用 PyQt/PySide/wxPython 可以开发出跨平台的应用程序。相比各个操作系统的原生编程语言，这对人力资源有限的开发团队很有吸引力。

1.3.5.4 数据端

数据从设备端收集上传到服务器端后，需要进行数据统计、集成、分析和可视化，才能够被企业所利用。无论是科学计算领域、金融分析领域还是大数据分析领域，Python 的市场份额都比嵌入式、桌面、服务器和移动端要大。

简单的数据收集和统计在各个环节中都存在，但是大数据分析需要配合足够的基础建设架构才能够实现，并通过 Hadoop/Storm/Lambda 实现批量和实时大数据分析。现在大多数大数据应用均依托 BAT/Bluemix/Azure/AWS 等具备大数据计算的云计算平台来实现，以降低硬件和运维成本。

1.3.6 标准、现状与未来

结合上面关于物联网的发展阶段和网络拓扑，就不难理解各行各业集体发力物联网领域的原因了。由于物联网的开发链条很长，而且容易形成生态，可以进行跨界整合，所以无论是传统大企业，还是专门领域的中小企业，都有足够的生存空间。

1.3.6.1 行业标准与私有标准

物联网发展了二十多年。其环节长，参与企业多，出现过各种标准和生态之间的争夺。将来这种合纵连横的状态只会更加激烈。尤其是互联网企业，已经颠覆了不少行业及行业标准。作为开发者和企业，需要仔细观察。笔者作为开发者，喜欢开源设计和行业标准，而不喜欢封闭的私有标准。

什么是行业标准？一个行业中所有供应商都自觉遵守执行的标准，即行业标准。

什么是私有标准？一个或者若干企业联手推出，公开或者不公开，采用行政强制，或者市场垄断、专利授权等方式进行推行的标准，都是私有标准。

为什么要澄清行业标准、私有标准的区别，就是想要区别企业标准、国家标准和国际标准。作为设计者来说，我们设计的产品和服务要服务于大众，而不是受限于某些技术垄断集团。有些技术集团打着国际标准和国家标准的旗号，企图实现垄断，这往往与技术无关，而与商业利益有关。

所以，开发者在具体选择标准的时候，需要做到：

- 不要迷信各类标准组织。要遵守行业标准，但是要留意私有标准组织的生态变化。
- 如果要建立生态，请公开标准，不要在标准上谋私利。
- 如果要进入某个生态，那么请遵守行业标准。

许多企业搞平台、生态、垂直整合、行业整合、水平整合。利用自己的市场、资金和技术垄断地位进行整合是这些企业的追求。但是现在的趋势是，企业已经无法垄断技术、资金，而市场也是千变万化的。作为开发者，我们应该重点关注：

- 设备组网、联网技术标准。
- 设备云平台，包括 IoT 的 PaaS/SaaS/CaaS 各类云服务的标准化。

1.3.6.2　Gartner 物联网研究报告

Gartner 是一家著名的 IT 研究和分析顾问公司，对于 IT 行业曾经做出过许多精准的预测。该公司针对物联网的发展趋势有过许多研究报告。原文网址请查看本章的延伸阅读部分。

笔者阅读之后发现，本书基本上或多或少都已经涉及了其所谓技术趋势。不过它的预测报告代表了投资界对于物联网的关注特点。所以，本书特地按照笔者的理解逐条稍加解释。

1．物联网安全

物联网的崛起为联网设备本身、平台与操作系统、通信方式，甚至是其所连接的应用系统，带来了各种新的安全风险及挑战。安全技术必须能保护联网设备与平台免受信息攻击与物理实体破坏。例如，将通信过程实施加密功能，以及解决“冒名物件”（impersonating things）、拒绝休眠（denial of sleep）耗尽设备电池等新型攻击方式。物联网安全状态将变得更加复杂，因为许多设备都使用了简易型处理器及操作系统，而没有使用高级安全方法。

有经验的物联网安全专家人数不多，而且目前只有来自不同厂商的零散的安全解决方案。随着黑客使用不同方式来攻击物联网设备及通信协议，2021 年以前将不断出现新的安全威胁。因此，需要长期部署的联网设备需要具备可以升级的软硬件设计，才能满足产品完整的生命周期需求。

现在的黑客活动目标明确，会针对特定的军事、商业、工业和个人目标进行攻击。针对物联网目标，黑客们会事先进行调研，扫描技术漏洞，取得设备控制权，并实现利益最大化。一

些著名案例如下：智能手机远程劫持，汽车和 ATM 攻击，通过 WebCAM 发动 DDoS 攻击等。

开源技术除了为企业带来利润，也是黑客们的主要技术来源。Python 作为主要的黑客语言之一，贡献了相当多的扫描工具和攻击工具。

笔者长期从事智能卡和 RFID 行业，虽然不是算法方面的专家，但却了解各个行业的安全需求。尤其对一些标准算法的实施有一定程度的了解，能够给出大致的解决方案。这些方案包括在设备设计领域中的 MCU/MPU 中可实施的对称、非对称算法，独立的安全认证 IC，内嵌安全模块的 SoC，在 FPGA 上实施的方案，TCP 及 UDP 通信加密，以及套接字服务器和安全 Web 的实施方案。

互联网是一个“黑暗森林”，适用于丛林法则。黑客攻防是一个“道高一尺魔高一丈”的领域。Gartner 将安全列入**第一考虑要点**值得所有物联网行业的同仁注意。实际上，只要是投入生产环境，就必须要注意这一点。就算提供只读数据的传感器，也需要反复评估是否在极端情况下产生副作用。本书第 4、9、10 章中会针对设备安全元组件、中间件、服务器 Web API 的安全算法以及日常运维的安全问题做一些介绍。

2．数据分析技术

物联网的使用者以各种方式利用设备收集信息，以便了解顾客行为、提供服务或改善产品，或者用来辨识及把握商机。不过物联网需要全新的分析方法、分析工具与运算法则，因为 2021 年以前数据容量将持续增加，物联网的需求可能会更加偏离传统分析技术。

物联网的真正业务模式取决于大数据分析。不过，单一设备累计的数据虽然容量巨大，却并不符合大数据多样性的定义。许多物联网领先企业其实也仅仅是保留积累数据，却无法利用数据，因为其数据类型太单一。Gartner 的说法对于“传统分析技术”的定义不够具体。笔者推测文中所指的非传统分析技术大概是大数据分析中的数据挖掘算法。数据分析与随后的数据交换和变现，所衍生的应用融合可以产生更加深远的意义。

比如收集用户使用医疗设备的频度和治疗方案（服用药物）之间的关系，是临床医学的分析手段。又比如，收集不同医疗设备的不同物理量（即生化指标）之间的关系可以用于分析病情的变化。物联网在这些方面均可以提供相关帮助。

由于在大数据分析领域的经验有限，因此，目前笔者主要基于设备云接入端提供简单的数据统计及数据可视化的方案。本书第 10 章简要介绍了笔者开发的服务器在数据端的应用。

3．设备管理

持续工作的重要设备需要管理与监测，这包括设备监测、固件与软件的更新、诊断、故障分析与回报、实体管理以及安全管理。物联网还给管理工作带来了很大的新问题，相关工具必须有能力管理并监测数千乃至数百万台设备。

笔者觉得这是物联网设备云的基础需求。监控数千还是数百万台设备对于架构成熟的系统来说不是最重要的问题。不过对于企业来说，凡事从小到大，选对适合当前规模的架构是非常

重要的。在工程上马初期强调未来的扩展，既不切实际，又浪费资源和开发时间。本书第 9 章中将介绍如何根据实际需求定制服务器架构。

4．低功耗短距离无线网络

物联网设备选择无线网络，必须在众多冲突的条件之间取得平衡，包括覆盖范围、电池续航力、频宽、密度、终端成本与运营成本。2025 年以前，低功耗短距离网络会成为无线物联网连接主流，普及程度将远超广域（wide-area）物联网网络。然而商业与技术上的权衡意味着将会有许多解决方案共存，不会由单一赢家，或者特定技术、应用程序或厂商生态系统独霸市场。

低功耗 WSN 网络协议四分五裂，实际上在可预见的将来依然无法统一。行业条块分割和局部垄断加剧了这个局面。应对的方式是采用标准接口将 WSN 设计成可替换的模块，构成一个通用的 WSN 网关，将其接入 TCP/IP（IPv6）网络。接入技术的标准化和高层应用抽象化是简化开发的方向。本书在第 5 章中介绍了这些连接标准，并在第 6 章及 9.3 节中介绍了如何构建、使用 Python 虚拟机，并基于 Linux/C/C++/Java 现有平台，通过 Python 来实现物联网网关应用。

5．低功耗广域网络

对于广覆盖、低频宽、电池长时间续航、低成本、高连接密度的物联网应用来说，传统的蜂窝网络无法满足其所要求的条件组合。广域物联网网络的远期目标，是通过全国性覆盖将数据传输率从几百 bps 提升到几十 kbps，同时电池续航力最多可达 10 年、终端硬件成本控制在 5 美元上下，而且能支持数十万台同时连接的设备。第一批低功耗广域网络（LPWAN）技术都是专有（proprietary technology）技术，但从长远来看，窄频物联网（NB-IoT）标准将成为这个领域的主流。

除了 4G/5G 蜂窝数据通信，Semtech LoRa 是目前可用的 LPWAN，同时 SigFox 为 LPWAN 的网络运营提供了标杆和参考。这两种网络与 IP 网络可以有机结合，形成局部区域的物联网运营商。不过这在国内需要政策支持。从目前来看，华为主推的 NB-IoT 将是运营商级别物联网的事实标准。此外，在 ISM 频段内使用 LTE 技术也是一种重要的技术方案。这方面的各类技术方案，竞争非常激烈。本书对于 LPWAN 的介绍主要集中在第 5 章，同时 9.3 节中介绍了 LoRaWAN 网络服务器对于网关设计的启发。

6．处理器

物联网设备所使用的处理器与架构决定了设备性能，例如安全性与加密功能、能耗技术，操作系统支持、固件更新能力，以及嵌入式设备管理代理程序（management agent）。在硬件设计方面，必须就各种功能、硬件成本、软件成本、软件升级能力等层面进行复杂的权衡考量。因此，要了解选择不同处理器隐含的技术与商业考虑，就必须具备深厚的技术能力。

许多创业者都说硬件“坑”多，这主要指的是硬件创业领域有着许多软件开发者不曾了解的技术、工艺和生产问题。而这些都需要经验积累。仅从技术角度看，选定一个平台，意味着

选择一个完整的开发生态。所谓生态包括：

- MCU 资源配置、供货、价格、版本、Bootloader 的选择；
- C/C++编译器限制和价格，JTAG 仿真器能力和价格；
- 中间件的成熟度；
- 工程的平均开发周期等。

这些变量都必须计算在内。许多企业认为在自己的专业领域内设备开发已经非常成熟了。但是网络将给设备开发带来新的变量和更多挑战。尤其是许多企业希望将设备的某些遗留技术带入物联网时代，这或许会给自己埋下一个“大坑”。本书对于设备端开发的介绍集中在第 4 章，希望能够让读者对相关平台和行业背景有个大体了解。

7．操作系统

Windows 或 iOS 之类的传统式操作系统（OS），均非针对物联网应用所设计。它们会消耗较多电力、需要高性能处理器，而且某些状况下缺乏硬实时特性。它们的存储器体量对小型设备来说过大，而且可能不支持物联网开发人员所使用的处理器。因此，为满足不同硬件配置与功能需求，目前市场已开发出各式各样的物联网操作系统。

笔者编写此书的目的之一就是基于开源操作系统和同一编程语言构建一个可重用的结构。Python 本身并不一定需要 RTOS，但是 RTOS 可以帮助简化系统的设计。同时，一些 Python 虚拟机底层也采用了某些 RTOS 内核。本书会在第 4 章简单介绍商业和开源的 RTOS，以及配套的中间件。

此外，硬件的大小核设计如果使用得当，在小核心中运行 RTOS，在大核心中运行 Linux+Python 也是很吸引人的设计，而且综合开发成本比单独运行一个操作系统要低。本书对于 Linux 的介绍贯穿在多个章节。

8．事件流处理

某些物联网应用会产生很高的数据传输率，这些数据必须进行实时分析。物联网应用系统往往每秒就会产生数万个事件，在某些电信或遥测（telemetry）案例中甚至会达到每秒数百万个事件。为解决相关需求，分布式流计算平台（DSCP）应运而生。这些平台通常会利用并行处理架构来提高数据传输和处理能力，以完成实时分析、模式识别等任务。

考虑到物联网的许多物理量实时分析并不需要很复杂的逻辑，因此笔者正在考察在服务器上利用各种硬件，如 GPU 加速、GPU 阵列，以及 OpenCL 之类的技术形成实时分析平台的可行性，还有客户对于这些技术的需求。此外，Lambda 等兼顾实时性的大数据平台也是考察标的平台。与大数据相关的介绍主要在第 10 章，而 GPU/FPGA 之类的应用主要在第 4 章介绍。

9．云计算平台

现在的物联网平台实在太多了，许多机构都宣称自己是“物联网平台”。

物联网平台能将物联网系统中诸多基础架构元件捆绑成单一产品，并提供几大类服务。

- 底层设备控制与营运：包括通信连接、设备监测、设备管理、安全管理、固件更新。
- 物联网数据与信息的采集、转换与管理。
- 物联网应用程序开发：包括事件驱动逻辑、应用程序设计、可视化、分析技术以及用来联结企业系统的转换器。

也有一些咨询公司将其归类为四大种类的“物联网平台”，具体划分如表 1-1 所示。

表 1-1　物联网平台功能定义

英文缩写	英语全称	译　　文	主要功能	主要参与方
DMP	Device Management Platform	设备管理平台	远程监控、设置调整、软件升级、系统升级、故障排查、生命周期管理等功能	端到端方案的一部分
CMP	Connectivity Management Platform	连接管理平台	连接配置和故障管理、保证终端联网通道稳定、网络资源用量管理、连接资费管理、账单管理、套餐变更、号码/IP 地址/Mac 资源管理	电信运营商
AEP	Application Enable Platform	应用使能平台	提供应用开发和统一数据存储两大功能的 PaaS 平台，架构在 CMP 平台之上	PTC/Cumolocity/Xively
BAP	Business Analytics Platform	业务分析平台	大数据分析、企业数据服务	人工智能和大数据分析服务供应商

这四种“物联网平台”侧重点不同，但是核心功能都是设备连接。本书并没有提供针对特定平台的应用介绍，主要在第 9 章中介绍如何基于 Python 构建自己的物联网设备云、Web 服务器，以及 Web API。

10．标准与生态系统

标准与生态系统并不属于技术范畴，但大部分终将具体化为应用程序编程接口（API）。标准与相关应用程序编程接口都会变得极为重要，因为物联网设备必须能够互通互联，而且许多物联网商业模式都仰赖不同设备与企业（机构）之间的信息分享。

未来市场将有诸多物联网生态系统崛起，而这些生态系统之间的商业与技术之争，也将主导智能家居、智慧城市与健康医疗等领域。制造产品的企业可能必须开发出变种版的产品来支持多种标准或生态系统；同时因为标准会持续演化，新的标准与相关应用程序编程接口也将崛起，制造产品的企业也要准备在整个产品的生命周期对其进行更新。

对于制造业和物联网企业来说，设计一个合适的硬件平台，让软件得以持续更新是一个必须要牢记的原则。即使在企业和产品生命周期完结后，也请抱着对社会和用户负责的态度将软硬件设计开放出来，这样产品的生命周期才会被无限延长。就像 Linksys WRT54G，2002 年生产，2004 年停产至今已有十余年，却依然是智能路由器的标杆平台。许多开源固件如 DDWRT、OpenWRT 都基于该平台提供新的服务，许多智能路由器依然受益于 WRT54G 的固件。

与此形成鲜明对比的是，许多智能硬件最初很新潮，但是过了一段时间就被人们淡忘了。越是新潮，被人遗忘得越快。本质上，智能硬件的商业逻辑和物联网设备的生命周期要求是相悖的。不是吗？人们需要能够默默地工作于周围的物联网。而许多智能硬件最多不过有些潮流感，或者短期内解决了某些问题；但如果这不是一种持续的需求，它们最多也就是玩具而已。

1.4 本章小结

本章主要介绍了物联网的核心价值、发展阶段、所需技术栈和相关产品，以便读者对物联网有个大致了解。

第 2 章

Python 语言基础

Python 的正确发音：/ˈpaɪθən/。用不太科学的中文标注法，发音是“派桑”，而非“派松”。Python 直接翻译为“蟒蛇”，其徽标如图 2-1 所示，为两条扭在一起的大蟒蛇。

图 2-1　Python 徽标

Python 易学易用，这是大家的共识。对于 C/C++/C#/Java 开发者而言，Python 的语法看上去非常熟悉。笔者本人几乎是读了 Python 的基本语法和一些例子之后，就立刻启动了一些工程开发。然而，这种急于求成的做法反而浪费了笔者的不少时间来重新造轮子。举几个例子：

- 用字符串列表来做二进制转换，事后发现 struct 就可以实现转换了。
- 二进制处理时，最初将二进制逐个转换类型后打印，事后发现 binascii 就可以实现了。
- 用 socket 做套接字服务器，事后发现有 Twisted 可以实现更多功能。
- 用 Twisted Web 做 Web 服务器，事后发现有 Klein 和 Cyclone 等更加完整的设计。
- 用 for 循环迭代来完成数据采集、类型转换、数据缩放和解复用，后来发现使用匿名函数以及高阶函数等编程技巧可以在一两行内实现以上任务。

诸如此类现象还在持续发生。不断发现自己还在重复造轮子，需要重构之前的代码。这种现象的主要原因在于笔者原有的从业经验中 C 语言的烙印太深，一时半会儿改不过来，还需要时间来体会 Python 实现更抽象编程模式的好处。

如果 Perl 语言推崇的哲学是，一项任务可以有多种（魔术般的）方法来完成，那么 Python 推崇的哲学就是只可以有一种方法来实现。这不代表 Python 代码只可以有一种方式来实现工程目的，而是意味着开发者会在开发过程中不断地尝试以最优化的代码去实现目标。

相信许多读者还是会犯同样的冒进错误，会尝试跳过本章去看后续章节。这没有问题，Python 的设计包容性很强，无论是面对过程、面向对象、面向切面，还是函数式编程，读者都有机会慢慢体会，慢慢进步，并不影响在实际工程中使用 Python。重构代码本身就是学习 Python

的最好经验，比任何教程的演示代码更接地气。

Python 的一大优点就是自带一大堆库，以及大量的第三方包。Python 程序员大部分时间是在找轮子，并评估使用这些轮子，而不是造轮子。所以做任何设计之前可以先行查看是否已有前辈贡献过代码。这些代码的主要来源如下。

- PyPI：https://pypi.python.org/pypi，在本书编写之际，共有 98 709 个软件包。
- GitHub：作为主流的 Git 软件库，有不少 Python 软件包。
- bitbucket：也有部分 Python 托管项目。
- Google code：目前为只读状态，但是一些比较久远的 Python 项目都托管在这里。
- sourceforge：它是许多需要单独安装的软件包的托管网站。

Python 简单易学，掌握其丰富的内置类型、数据结构、标准库和第三方软件扩展，可以让我们事半功倍。虽然只用最基础、最常见的数据类型也可以设计出物联网的设备和服务器，但许多时候采用 Python 内置数据类型更加高效。这一点，在嵌入式 Python 虚拟机和应用设计中尤为重要。作为标准 Python 的子集，嵌入式 Python 中舍去了许多内置类型，针对嵌入式定制了数据类型，并有不同的限制。

Python 最权威的文档就在 Python 基金会网站中，其事无巨细地记录了从 Python 的早期版本到最新版本的变更和各类教程。阅读完这些资料的确需要花很多时间，而且有些文档也需要具备 Python 开发经验才可以读懂。所以，笔者找到了一个相对精简的资源——*The Python Standard Library by Example* 的作者 Doug Hellmann 有一个专门的网站 Python Module of The Week。推荐大家将这两者互为补充阅读。

另外还有一本书 *Python Cookbook*，针对 Python 的各类进阶用法，有一个比较完整的归类和实例。

国内也有不少 Pythoner 提供了很好的在线教程，如廖雪峰的 Python 2/Python 3 教程，其行文风格生动活泼，值得推荐，请在延伸阅读中寻找相关网址。

受限于篇幅，本章主要介绍 Python 语言本身（不包括第三方库）在物联网编程中会使用到的基础以及进阶知识。本书假设读者有一定 C/C++/C#/Java 等编程语言的经验。Python 相对这些语言更像伪代码，所以对于有过编程经验的读者来说很容易理解。

从编程角度来看，物联网编程就是数据在采集、转换、传输、处理、存储、融合、可视化各个环节中所用的各类方法。Python 与物联网相关的部分知识主要包括：

- Python 与嵌入式和 C/C++之间的数据类型、数据结构的异同及相关接口；
- Python 中的数据类型、数据结构与高级编程范式；
- Python 中数据类型的灵活使用，以节省资源，并提高代码性能；
- Python 各种内置类型之间的转换和操作；
- Python 与物联网密切相关的标准库如数学、网络、加密、网络操作等；

- 某些必要的 Python“语法糖”;
- Python 与其他语言如 C 和 Java 之间的接口。

其他如操作系统、文件、GUI、调试等都放到其他章节中，采用标准库和流行第三方库以实例说明。

2.1　Python 的由来与特征

下面介绍 Python 的一些基本信息。

2.1.1　概述

Python 是一种面向对象、解释型计算机程序设计语言，由荷兰人 Guido van Rossum 于 1989 年发明，其第一个公开版本发行于 1991 年。Python 是纯粹的自由软件，源代码和解释器 CPython 遵循 GPL（GNU General Public License）协议。Guido 非常喜欢 20 世纪 70 年代的一部英国肥皂剧《巨蟒飞行马戏团》（*Monty Python's Flying Circus*），所以选中 Python 作为该语言的名字。

Python 语法简洁清晰，其特色之一是强制用空白符（white space）作为语句缩进。通过强制程序缩进（包括 if、for 和函数定义等所有需要使用模块的地方），Python 使得程序更加清晰和美观。

Python 具有丰富和强大的库。它常被称为“胶水语言”，能够把用其他语言制作的各种模块（尤其是 C/C++）很轻松地整合在一起。常见的一种应用情形是，使用 Python 快速生成程序的原型，然后对其中有优化要求的部分，用更合适的语言改写，比如 3D 游戏中的图形渲染模块，性能要求特别高，就可以用 C/C++重写核心代码，封装为 Python 可以调用的扩展类库。

由于 Python 语言的简洁、易读以及可扩展性，在国外用 Python 做科学计算的研究机构日益增多，一些知名大学已经采用 Python 讲授程序设计课程。例如卡耐基梅隆大学的编程基础、麻省理工学院的计算机科学及编程导论就使用 Python 语言讲授。众多开源的科学计算软件包都提供了 Python 的调用接口，例如著名的计算机视觉库 OpenCV、 三维可视化库 VTK、医学图像处理库 ITK。而 Python 专用的科学计算扩展库就更多了，例如经典的科学计算扩展库：NumPy、SciPy 和 matplotlib，分别为 Python 提供了快速数组处理、数值运算以及绘图功能。因此，Python 语言及其众多的扩展库所构成的开发环境十分适合处理实验数据、制作图表，甚至开发科学计算应用程序。

2.1.2　设计定位与哲学

Python 的设计哲学是“优雅”、“明确”、“简单”。因此，Perl 语言中“总是有多种方法来做

同一件事”的理念在 Python 开发者中通常是难以忍受的。Python 开发者的哲学是“用一种方法，最好是只有一种方法来做一件事”。所以，虽然 Python 依然有多种方式来实现同一工程目的，但作为 Python 程序员，往往需要不断地重构代码以追求最佳方案。

Python 开发者应尽量避开不成熟或者不重要的优化。一些针对非重要部位的加快运行速度的补丁通常不会被合并到 Python 内，所以很多人认为 Python 运行速度很慢。不过，根据二八定律，程序中的大部分代码对速度要求不高。在某些对运行速度要求很高的情况下，Python 设计师倾向于使用 JIT 技术，或者使用 C/C++语言改写这部分程序。

Python 是完全面向对象的语言，函数、模块、数字、字符串都是对象，并且完全支持继承、重载、派生、多继承，有益于增强源代码的复用性。Python 支持重载运算符和动态类型。

虽然 Python 可能被粗略地分类为“脚本语言”（script language），但实际上一些大规模软件开发计划如 Zope、Mnet 及 BitTorrent 均使用了 Python。Python 的支持者较喜欢称它为一种高级动态编程语言。Google 也广泛地使用它。在 Google 推出 Golang 之前，该公司聘用了 Python 之父 Guido，并一度成为严重偏爱 Python 语言的互联网公司。

此外，Python 也是许多黑客偏爱的编程工具，在网络安全、报文分析和渗透领域经常看到 Python 脚本。

2.1.3 优点与缺点

任何语言都有自己的特点，Python 的优点和缺点都很鲜明，所以经常被拿来与其他流行的编程语言进行比较。

2.1.3.1 优点

Python 有如下优点。

- 语法简单。

  Python 是一种代表简单主义思想的语言，号称是可以执行的伪代码。阅读一个良好的 Python 程序就感觉像是在读英文一样。它使开发者能够专注于解决问题而不是去搞明白语言本身。
- 容易学习。

  Python 极其容易上手，这与其语法简单、文档完整有关。
- 速度较快。

  Python 的底层是用 C 语言写的，很多标准库和第三方库也都是用 C 语言写的，运行速度较快。关于这一点，不同语言爱好者的评价是不同的。编程语言之间的运行速度竞争是动态的，在 LLVM 的推动下，许多编程语言的运行速度都得到了大幅度提升。
- 免费开源。

Python 是 FLOSS（自由/开放源码软件）之一，使用者可以自由地复制、查阅、移植、分发和整合在自己的软件中。

- 高级语言。
  用 Python 语言编写程序的时候无须考虑诸如如何管理内存一类的底层细节。Python 与其他高级语言相比，还可以实现更加抽象的编程范式。
- 可移植性。
  Python 已经被移植到许多平台上。这些平台包括 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE、PocketPC、Symbian 以及 Google 的 Android 平台。CPython 本身没有针对移植优化，移植过程相对复杂。
- 解释性。
  Python 语言写的程序无须编译成二进制代码，可以直接从源代码运行程序。在计算机内部，Python 解释器把源代码转换成被称为字节码的中间形式，然后再把它翻译成计算机使用的机器语言并运行。这使得使用 Python 更加简单，也使得 Python 程序更加易于移植。
- 面向对象。
  Python 既支持面向过程编程，也支持面向对象编程。在“面向过程”的语言中，程序是由过程或可重用代码的函数构建起来的。在“面向对象”的语言中，程序是由数据和功能组合而成的对象构建起来的。
- 可扩展性。
  如果需要提升关键代码性能，或者保护代码的知识产权，可以用 C 或 C++编写该部分代码，然后在 Python 程序中调用它们。
- 可嵌入性。
  可以把 Python 嵌入 C/C++程序，为用户提供脚本功能。
- 标准库与第三方库功能丰富。
  Python 标准库确实很庞大，它可以处理各种工作。除标准库以外，还有许多其他高质量的第三方库，可以完成特定领域的复杂任务。
- 代码规范。
  Python 采用的强制缩进方式使得代码具有较好的可读性。

2.1.3.2　缺点

对于其他语言的拥趸来说，Python 存在以下缺点。

- 单行语句和命令行输出受限制。
  很多时候不能将 Python 程序连写成一行，如 import sys;for i in sys.path:print i。必须将程序写入.py 文件中运行。而 Perl 和 awk 就无此限制，可以较为方便地在 shell 下完成简单程序。
- 语法独特。
  Python 采用强制缩进来区分语句关系的方式还是给很多初学者带来了困惑。即便是很有经验的 Python 程序员，也可能陷入陷阱当中。最常见的情况是 tab 和空格的混用会导致错误，而这是用肉眼无法区分的。不过，选用合适的编辑器可以解决此种情况。
- 运行速度慢。
  与 C 和 C++相比，Python 运行速度较慢。其与有 JIT 加速的各类语言相比也有一定差距。PyPy 加速也可以归类于 JIT，可以帮助 Python 实现加速。

2.2 Python 与物联网开发

由于预见到物联网的需求将十分旺盛，因此近年来许多大学纷纷开设了物联网专业，而不少学生则表示学得太杂。不过，这符合物联网的特性：多中心，分布式，技术庞杂，涉及面广，每一门技术都需要开发者钻研多年。

本书希望从开发实务角度，通过 Python 这一“胶水语言”，从物联网的各个开发环节（IC 数字逻辑、连接性、硬件平台、节点设备固件、Bootloader、网关硬件平台、网关软件、设备云服务器、大数据分析、移动 APP、桌面 APP 等）为切入点，提供一些简单有效的技术方案供读者选择；同时利用 Python 解决日常的开发痛点，降低学习门槛，缩短系统开发周期，加快迭代速度。

Python 拥有一个强大的标准库。Python 语言的核心只包含数字、字符串、列表、字典、文件等常见类型和函数，而由 Python 标准库提供了系统管理、网络通信、文本处理、数据库接口、图形系统、XML 处理等额外的功能。Python 标准库命名接口清晰、文档良好，很容易学习和使用。

Python 标准库的主要功能如下：

- 文本处理，包含文本格式化、正则表达式匹配、文本差异计算与合并、unicode 支持，以及二进制数据处理等功能。
- 文件处理，包含文件操作、创建临时文件、文件压缩与归档、操作配置文件等功能。
- 操作系统功能，包含线程与进程支持、I/O 复用、日期与时间处理、调用系统函数、日志记录等功能。
- 网络通信，包含网络套接字，以及 SSL 加密通信、异步网络通信等功能。

- 网络协议，支持 HTTP、FTP、SMTP、POP、IMAP、NNTP 等多种网络协议，并提供了编写网络服务器的框架。
- W3C 格式支持，包含 HTML、SGML、XML 的处理。
- 其他功能，包括国际化支持、数学运算、HASH、Tkinter 等。

Python 社区提供了大量的第三方模块，覆盖科学计算、Web 开发、数据库接口、图形系统等多个领域，并且大多成熟而稳定。第三方模块可以使用 Python 或者 C 语言编写，这使得以 Python 或 C++编写的程序能互相调用。借助标准库的大量工具以及调用其他语言程序的能力，Python 已成为一种强大的整合其他语言与工具的胶水语言，具备快速构建应用原型的能力。本书将在物联网的以下开发环节中介绍 Python 的应用。

- IC 设计与原型验证：采用 Python 进行数字和模拟电路的建模、仿真、测试和波形输出。
- MCU 固件开发：采用嵌入式 Python 虚拟机进行固件设计和系统定制。
- MPU/SoC 固件开发：嵌入式和标准 CPython 的交叉编译，在 Linux SBC 上设计 Python 网关。
- 桌面应用：采用 wxPython/PyQt/PySide 和 Web GUI 构建桌面应用。
- 移动 APP：采用 QPython/Kivy 和 Web+PhoneGap 设计移动 APP。
- 服务器：采用 Twisted/Cyclone/Tornado 开发高并发设备接入服务器，采用 Django/Flask 实现 Web 应用快速迭代。
- 数据分析：针对科学计算、工程计算、金融分析以及数据统计、分析、挖掘的 Python 平台与工具包。
- 其他工具：渗透在开发流程和应用各个环节的各类小工具、虚拟仪器、文档操作、配置管理、本地化、软件测试、媒体处理、报文分析、3D/VR 等多方面的辅助开发工具。

编写本书收集资料的过程，也让笔者开阔了视野：Python 原来还有这么多用法。

Python 物联网思维导图

为了表达 Python 在物联网中的应用场景，笔者特地利用“百度脑图”绘制了一个思维导图。由于该图过于复杂，请扫描以下二维码查看该思维导图。

2.3 获取 Python 资源

由于国内的网络环境，Python 主站有时候无法访问，所以，笔者特地整理了一下国内可以访问到的 Python 资源。

2.3.1 Python 主程序

如果无法从官网下载 Windows 版本的 Python 主程序，请使用 360 或者百度软件下载安装，缺点是无法挑选到特定版本。Linux/UNIX 系统中大多已经默认安装 Python。笔者推荐安装 64 位 Python。虽然许多 Python 2.7 的库依然仅仅支持 32 位，但是一些与大数据等有关联的 Python 库都已经不再支持 32 位。

除非你的项目需要在 32 位平台上运行，比如 Windows XP，否则还是使用更新的 64 位操作系统和 Python 版本吧。

2.3.2 Python 文档

安装了 Python 之后，在 Python 安装路径中可以看到 Doc 文件夹。里面有相关的 Python 文档，Windows 版本提供了 chm 文档。Linux 中则可以启动 pydoc 文档服务器后，通过浏览器查找文档。

Python 文档很齐全，包括：

- 官方教程，介绍 Python 语言和系统的基本概念和功能；
- 语言参考，Python 语义与语法定义；
- 标准库，Python 自带的标准对象（内置类型，内置函数）和模块的说明；
- 扩展和嵌入，如何扩充 Python 的功能；
- FAQ，常见问题；
- HOWTO，一些常见任务的解决方案。

此外，感谢国内的 Pythoner，即使官网无法访问，还可以找到 Python 文档的镜像站点和翻译站点：

- http://docs.pythontab.com/；
- http://www.pythondoc.com/；
- http://python.usyiyi.cn （Python 文档中文版）。

2.3.3 Python PyPI

PyPI 的全称是 Python Package Index，即 Python 软件包索引。在国内偶尔无法访问官方 PyPI 服务，可以采用 PyPI 镜像来访问。

例如：

- http://pypi.douban.com/simple/，豆瓣网镜像。
- http://mirrors.aliyun.com/pypi/simple/，阿里云镜像。

在教育网网内请使用各个高校的 PyPI 镜像。

以安装 Twisted Python 包为例，通过镜像安装 Python 扩展包的命令是：

```
pip install twisted -i http://pypi.douban.com/simple/
```

豆瓣网在 pypi.douban.com 上没有配置 HTTPS，只是 HTTP。可以使用以下方式信任该节点：

```
pip install twisted --trusted-host pypi.douban.com
```

每次都输入镜像节点有些啰唆，可在 Python 系统配置中将其设置为默认配置。

Windows

```
%HOMEPATH%\pip\pip.ini
```

Linux

```
~/.pip/pip.conf
```

```
[global]
timeout = 6000
index-url = http://pypi.douban.com/simple/
[install]
use-mirrors = true
mirrors = http://pypi.douban.com/
```

但笔者总是优先使用主站下载，万不得已才使用镜像节点。

有些 Python 包采用独立安装包，比如 wxPython 等，需要自行通过 sourceforge 及其镜像站点下载。

PyPI 安装错误原因与对策

现在 Python 推荐从 PyPI 及其镜像节点安装软件包。但是许多时候却会报错，其原因多种多样：

- 镜像节点更新不及时。
- 操作系统依赖，由于 32/64 位等原因，缺乏对应版本。
- 开发商并没有向 PyPI 提交软件。

所以，终极方案是通过源码安装。一般可以寻找对应的安装包，或者下载源码压缩包，解压缩后运行其中的 setup.py 即可以完成安装。

有些 Python 包采用独立安装包，比如 wxPython 等，需要自行通过 sourceforge 及其镜像站点下载。

例如：

- http://pypi.douban.com/simple/，豆瓣网镜像
- http://mirrors.aliyun.com/pypi/simple/，阿里云镜像

2.4 Python 解释器运行环境

在 Windows 中，安装后 py 后缀即被关联到 Python 主程序。其安装路径为：

- C:\Python27
- C:\Python3

在 Linux 中，Python 的安装路径为：

- /usr/local/bin/python
- /usr/bin/python

可以采用 which python 来定位 Python 软件所在路径。

2.4.1 REPL 交互模式

直接运行 python，不带任何参数，可以进入 Python 的 REPL 交互式环境。REPL（Read-Evaluate- Print-Loop）是通过命令行逐句执行语句的方式。它是小规模学习和评估一种语言的最直接方式。包括 Python 在内，还有许多编程语言如 Lua 等都具备 REPL 接口。

退出 REPL 交互方式有三种方法：

- 输入“exit()”；
- 输入“quit()”；
- 按组合键退出（Ctrl-D：Linux，Ctrl-Z：Windows）。

```
C:\>python
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 20:40:30) [MSC v.1500 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

2.4.2 直接运行与模块运行

除了 REPL 交互模式，无论是 Windows 还是 Linux 中，都可以在命令行中利用 Python 解释器来运行 Python 脚本或模块。假设用户脚本名称为 script.py，那么还要分为两种：直接运行和模块运行方式。

2.4.2.1　直接运行

将文件名作为参数传给 python，即可以运行指定脚本。

```
python demo.py args
```

2.4.2.2　模块运行

也可以将用户代码作为模块运行。

```
python -m demo args
```

两者的差异在于 sys.path 的值不同。假设我们有脚本 runmethod.py：

```
import sys
print sys.path
```

分别以直接运行与模块启动方式运行。

```
python runmethod.py

['C:\\PHEI_Book\\source_code\\python-basic\\runmethod', 'C:\\Windows\\system32\\
python27.zip', 'C:\\Python27\\DLLs', 'C:\\Python27\\lib', 'C:\\Python27\\lib\\pl
at-win', 'C:\\Python27\\lib\\lib-tk', 'C:\\Python27', 'C:\\Python27\\lib\\site-p
ackages', 'C:\\Python27\\lib\\site-packages\\FontTools', 'C:\\Python27\\lib\\sit
e-packages\\win32', 'C:\\Python27\\lib\\site-packages\\win32\\lib', 'C:\\Python2
7\\lib\\site-packages\\Pythonwin', 'C:\\Python27\\lib\\site-packages\\wx-3.0-msw
']

python -m runmethod.py

['', 'C:\\Windows\\system32\\
python27.zip', 'C:\\Python27\\DLLs', 'C:\\Python27\\lib', 'C:\\Python27\\lib\\pl
at-win', 'C:\\Python27\\lib\\lib-tk', 'C:\\Python27', 'C:\\Python27\\lib\\site-p
ackages', 'C:\\Python27\\lib\\site-packages\\FontTools', 'C:\\Python27\\lib\\sit
e-packages\\win32', 'C:\\Python27\\lib\\site-packages\\win32\\lib', 'C:\\Python2
7\\lib\\site-packages\\Pythonwin', 'C:\\Python27\\lib\\site-packages\\wx-3.0-msw
']
```

读者可以发现直接运行方式会将脚本所在路径加入 sys.path，而模块运行方式则是空的。这对用户自己的其他模块导入方式有很大影响。如果采用直接运行的方式，则用户模块必须放置在同一文件夹内。如果采用模块运行方式，则其他的模块可以放置在单独的文件夹，即包内。反之，都会报错。

2.4.3　脚本文件直接运行

每次运行用户脚本，如果都前置输入 python 命令，显得很多余。在大多数操作系统中，都

支持使用脚本文件名直接运行。即：

```
myscript.py args
```

在 UNIX/Linux 中，Python 源码运行需要增加 shebang 声明。

```
#!/usr/bin/env python
print "hello world"
```

shebang（#!）来自 UNIX shell 脚本。在 UNIX/Linux 中，如果脚本文件中没有（#!）声明，那么 Linux 在执行该脚本时，会使用当前默认的 shell 去解释这个脚本（即$SHELL 环境变量所定义的解释器）。也就是说，将其文件中的 Python 代码作为 shell 脚本进行执行。如果（#!）之后的程序是一个可执行文件，那么执行这个脚本时，它就会把文件名及其参数一起作为合并后的参数，并将文件内容作为代码传给该执行文件去（解释）执行。

在 Windows 中，在安装了 Python 程序之后，Windows 会自动关联.py 和 Python 主程序。所以，也可以直接运行 Python 源文件。直接输入 script.py 等效于 python script.py。由于 Windows 关联是操作系统全局有效的，所以在多个版本 Python 时需要注意究竟是哪个版本 Python 关联了.py 后缀。

2.4.4 源程序文字编码与结束符

Python 源程序也是文本文件。当文件中有非英文字符串时，必须在文件头部定义文字编码。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

具体的做法是在 shebang 后注明编码，并将该文件按照所声明的编码进行保存。-*-分隔符不必严格遵守，实际上这一行只需要包含注释符号“#”、“coding”和编码如“utf-8”即可。分隔符可以自由定义。

除非在特定情况下，否则请不要使用 GB2312 等编码来保存包含中文字符串的代码。一般来说，为了兼容英语并支持多国语言，将其指定为 UTF-8 编码可以一劳永逸。文件保存时的编码格式必须与定义一致，否则会出现乱码或者直接报错。保存文件时请选择 UTF-8，但不要选择带 BOM（Byte Order Mark）标记的格式。在 Linux 下可以使用 file 命令来检测文件类型和编码，并使用 iconv/enconv 进行编码转换。

```
# file scheduler.py
scheduler.py: a python script, ASCII text executable
```

如果代码要做到 Windows/Linux 跨平台，请务必使用 Linux 行尾结束符；否则，Windows 下编辑的源码在 Linux 中运行，操作系统会报错，而且提示内容与行尾结束符无关。

```
# ./CheckRealtimeDevices.py
: No such file or directory
```

遇到此类报错，只要将其变成 UNIX 结束符就没有问题了。有关其他运行选项，请在命令行下输入 python –help。查看 Python 程序的运行选项。

2.5　Python 类型与语法

Python 官方提供的文档 *The Python Library Reference, Release 2.7.11* 是一个最大的 Python 文档，有 1 300 多页。基于 Python 语言本身的特性，还将内置数据类型也归于该文档。

所谓 Python 库（Python Library）包含了若干不同类型的组件。

首先，它包括数据类型如数值和列表类型，这通常被认为是语言内核的一部分。针对这些类型，Python 内核做了形式上的定义，并做了一些语义上的限制，但是却没有定义其语义实现。换而言之，语言内核没有定义拼写和操作符优先级之类的语法属性。

其次，它包含了内置函数和异常处理，即无须使用 import 语句就可以使用的 Python 代码。此类代码不是核心语义，而在库里定义。

最后，是我们通常意义上的库，由一堆模块组成。可以用不同方式来归类这些库：有些使用 C 语言编写，并内置于 Python 解释器中；有些采用 Python 编写，并以源码形式导入；有些模块提供的接口专门为 Python 定制，如打印 Stack Trace；有些模块针对特定操作系统或特定硬件定制；还有针对特定领域开发的，如 Web。

标准库部分体现了 Python Battery Included 的特点，也是 Python 成为一种强大编程语言的原因。

此外，本书基于 Python 2.7 进行介绍。没有直接从 Python 3 开始介绍的原因，是因为许多历史悠久的相关框架还没有迁移到 Python 3。就目前来看，使用 Python 2.7 依然是一个比较合适的方案。当然，如果日后读者需要自行开发新应用，基于 Python 3 进行开发是值得推荐的。所以，本书会加入一些兼容性设计内容介绍。

Python 语言与 C 谱系语言有许多方面都很类似。但作为动态类型语言，其内置类型和实现方法却有些区别，并产生了使用上的差别。接下来，我们先从最基本的内置类型开始。

2.5.1　动态类型

在计算机语言中，变量可以被赋予各种类型的数据。在 Python 中，和其他语言类型一样，也采用“=”作为赋值语句。一个变量可以反复赋值，而且数据类型可以不断地变化。这就是所谓的动态类型语言。

```
s = 1234
s = 'wxyz'
```

Python 的动态类型特性和 C/C++/Java 等静态编译语言存在很大不同。在 C 语言中，任何变量均需要事先声明，用于从系统中申请合适的 RAM/ROM 存储空间。如果把不同类型的数据重新赋予变量，编译时就会报警甚至报错。解决方式是采用强制类型转换。

```
unsigned char t = 0;
t = 10000; // 错误，强制赋值会失去高位数值
```

2.5.2 传值与传引用

变量在内存中的表达主要有两种：传值或传引用。不同语言之间存在很大差异。

Python 和 C 语言中的变量在内存中的表达也不同。在 C 语言中，变量名在声明类型后，往往被赋予内存或者寄存器中的某个地址开始的空间。

```
unsigned char v1 = 0; # 假设 V1 在 RAM 0x1000，保存整数 0
unsigned char v2 = 1; # 假设 V2 在 RAM 0x1010，保存整数 1
```

如果要互换 v1/v2 的内容，则需要借助第三方变量 t 做缓存。

```
unsigned char t;
t = v1;
v1 = v2;
v2 = t;
```

但是在 Python 中，我们却可以这样做：

```
>>> x = 123
>>> y = 456
>>> y,x = x,y
>>> y,x
(123, 456)
>>>
```

为什么它们不需要中间缓存变量呢？这里我们需要提前使用 Python 内置函数 id 来研究它们之间的差异。该函数专门用于打印对象的内存地址。

```
>>> x = 123
>>> id(x)
36275952L
>>> y = 456
>>> id(y)
41956536L
>>> y,x = x,y
>>> id(x)
41956536L
```

```
>>> id(y)
36275952L
>>> y,x
(123, 456)
>>> z = 123
>>> id(z)
36275952L
```

最初赋值时，变量 x 指向地址：36275952L，而变量 y 指向：41956536L。变量互换赋值后，y 指向位置和 x 指向位置互换了，y 和 x 指向的整型变量也跟着变化了。如果对 z 赋予与 y 变量一样的整数，发现 z 变量指向位置居然与 y 指向位置一样！

这是因为 Python 变量在内存中的表达与 C 语言是不同的。

- 在 x = 123 赋值语句中，Python 创建了变量 x 和整数 123，并把 x 指向整数 123 的地址：36275952L。
- 在 y = 456 赋值语句中，Python 创建了变量 y 和整数 456，并把 y 指向整数 456 的地址：41956536L。

在互换赋值执行后，x 指向 41956536L，y 指向 36275952L，由于整数并没有保存在 x 和 y 中，而是另外分配的整数‘123’和‘456’的存储空间，所以无须中间变量来交换。最后，z = 123 执行后，Python 创建了变量 z，还把现有整数 123 的地址也给了 z。所以现在 y 和 z 指向了同一个整数。这就是 C 语言中传值与 Python 传引用的最大区别。

回看自己最初的 Python 代码，傻傻地在 Python 中大量使用中间变量实现变量交换，笔者不禁莞尔。依此类推，Python 赋值语句还可以更加灵活。所以笔者特地安排这一章节的目的就是让熟悉 C 语言编程习惯的读者意识到，既有的旧经验、旧习惯可能在 Python 编程过程中需要逐渐抛弃。

如果读者有 Java 的开发经验，可以和 Java 对比一下：Java 有原始类型（primitive），也有引用类型（或复杂类型），其中原始类型是放在栈空间的，而复杂类型则是“引用”在栈中，对象本身放在堆中；JavaScript 采用了类似的类型分类方法；而 Python 则走得更远，所有内置类型都是对象，都采用复杂类型存储模式。笔者在学习 Java/JavaScript 时，经常弄混淆传值和传引用的区别。到了 Python 这里就简单了，都是传引用。而 C 语言里都是传值。当然，使用指针可以实现传引用，但这是另外的故事了。

通过 C/Java/Python 三者的对比，就可以发现编程语言的不断演化，也可以促使读者仔细对比这些编程语言间的区别。Python 这样做固然有好处，但却要付出性能方面的代价。

2.5.3　数据类型

计算机能够处理不同类型的数据，包括数值、文本、图形、音频、视频和任何二进制数据。

与贴近机器硬件的 C 语言相比，Python 能够提供的数据类型要丰富灵活得多，其内在的数据结构也可以满足更加高级而复杂的程序设计需求。这些数据类型由内置类型（built-in types）和集合类型（collections）提供。

2.5.4 内置类型

数据类型在数据结构中的定义是一个值的集合以及定义在这个值上的一组操作。数据类型的出现是为数据划分存储空间。

通常在学习 C 的时候，我们知道 C 语言有如下若干原始（primitive）数据类型。

- 基本数值类型：short/int/long/float/double；
- 字符类型：char；
- 构造类型：数组/struct/union/enum；
- 指针类型；
- 空类型：NULL。

如果断章取义地去看 Python 教程，许多人就会有一个疑问：Python 究竟有多少种内置类型（built-in types）？好像文档里的“types”太多了吧！具体有多少类型，请在 REPL 下尝试一下：

```
>>> import types
>>> dir(types)
['BooleanType', 'BufferType', 'BuiltinFunctionType', 'BuiltinMethodType', 'ClassType
', 'CodeType', 'ComplexType', 'DictProxyType', 'DictType', 'DictionaryType', 'Ellips
isType', 'FileType', 'FloatType', 'FrameType', 'FunctionType', 'GeneratorType', 'Get
SetDescriptorType', 'InstanceType', 'IntType', 'LambdaType', 'ListType', 'LongType',
 'MemberDescriptorType',
 'MethodType', 'ModuleType', 'NoneType', 'NotImplementedType', 'ObjectType', 'SliceT
ype', 'StringType', 'StringTypes', 'TracebackType', 'TupleType', 'TypeType', 'Unboun
dMethodType', 'UnicodeType', 'XRangeType', '__all__', '__builtins__', '__doc__', '__
file__', '__name__', '__package__']
```

或者采用在线帮助来查看：

```
>>> import types
>>> help(types)
```

Python 2.7.11 大约有 37 种类型；Python 3 的实现更为复杂，而且还有些许变化。推荐查看 /usr/lib/python3.2/types.py 源文件。

首先，Python 没有 C 语言基础（primitive）数据类型的概念；其次，各种“types”术语会让读者糊涂。使用过 Python 一段时间后，笔者才体会到这背后的原因：

- Python 的所有数据类型都被封装在对象类中。
- Python 的所有东西都是可以访问的，包括编写的代码。

- Python 实际上不包括整数这样的简单类型，而是创建整数对象，并将新的对象引用赋值给变量。
- Python 是动态类型语言，并不需要使用前声明变量类型以在编译时分配存储器，变量类型都基于对象，可以在运行时改变。
- Python 的类型分类方式很多，原因是不同内置类型具备不同特性，所以采用不同划分方法可以划入不同的“类型”。

虽然是动态语言，大多数变量使用前无须声明，但一些序列类型依然需要使用前声明，以便让系统了解需要多少序列数据对象。由于整数在 Python 中也是对象，所以在嵌入式中显得特别耗费资源。为此，许多嵌入式版本 Python 还是针对不同内置数据类型做了不同程度的“降维”处理，将其作为与 C 语言相对应的 byte/word/long word 处理，以节省 RAM 和 ROM 资源。

综上所述，Python 的内置类型非常丰富，其文档中有一个长长的内置数据类型清单。在其官网文档中，Python 2.7.11 的主要内置数据类型分类有：

- 数值（numeric）；
- 序列（sequence）；
- 映射（mapping，键值对）；
- 文件（file）；
- 类（class）；
- 实例（instance）；
- 异常（exception）。

C 语言的基本数据类型可以映射到 Python 的数值类型和序列类型中的一部分。如果从 Python 解释器运行时可使用的类型来划分的话，可以分为四个类型。

- 简单类型：int、float、long、complex、bool；
- 容器类型：包含其他类型对象的类型，如 tuple、string、unicode、list、set、frozenset、dictionary；
- 代码类型：Python 程序本身；
- 内部类型：执行期间使用。

在内置类型中，还**必须**根据对象是否可进行修改和变化，分为可变类型（mutable，词根为 mut）和不可变类型（immutable）。此外还有迭代器（iterator）、生成器（generator）类型等。

某个类型如果采用不同划分方法，可以划归为不同类型。例如：tuple，元组，属于不可变类型、序列类型、容器类型，也是可迭代类型。这好比从不同角度对马进行分类：马是动物，是奔跑类动物，为草食类、奇蹄类。

2.5.5 内置类型的普适操作

所有内置类型都可以进行比较、测试真值、转换字符串操作。

2.5.5.1 真值测试

任何对象都可以做真值测试（Truth Value Testing），并用于 if 和 while 循环条件判断以及布尔操作的操作数中。以下数值可以理解为 False（假）。

- None；
- False；
- 任意数值类型的 0 值，如：0，0L，0.0，0j；
- 任意空序列类型：如：''，()，[]（即空字符串、元组、列表）；
- 任意空的映射类型：如{}（即空的集合）；
- 用户定义的类，如果类定义了__nonzero__或__len__方法，而且这些方法返回零值，或布尔量 False。

其他的均为 True（真）。所以大多数类型的对象都是真的。除了布尔操作符 or/and 返回其中一个操作数，布尔操作和内置函数总返回 0/False，1/True。

2.5.5.2 布尔操作

布尔操作包括：and，or，not。其操作参见表 2-1。

表 2-1 布尔操作

操　作	结　果	备　注
x or y	如果 x 为假，返回 y；反之返回 x	短路操作符，第一参数为假才计算第二参数
x and y	如果 x 为假，返回 x；反之返回 y	短路操作符，第一参数为真才计算第二参数
not x	如果 x 为假，返回 True；反之返回 False	not 优先级最低，not a==b 等效于 not (a==b)

为了性能加速，Python 采用了这种短路操作符。但是在许多其他语言的设计中，是先计算所有操作数后才做布尔操作的。所以，需要额外注意：最好不要在布尔操作中添加进行变量修改的操作。

2.5.5.3 比较操作

比较操作如表 2-2 所列，适用于所有对象。而且 Python 的比较操作比 C 简洁。可以使用：

```
a < b < c < d
```

而不需要采用 C 语言中那样写：

```
(a < b) && (b < c) && (c < d)
```

注意　上述比较操作依然存在短路操作。

表 2-2　比较操作

操　　作	含　　义	备　　注
<	小于	
<=	小于或等于	
>	大于	
>=	大于或等于	
==	等于	
!=	不等于	不要再使用<>
is	对象 id 相同	
is not	对象 id 不同	

尽管比较操作适用于所有内置类型，但是不同类型的对象比较，大多数是不等的。类的实例之间比较采用其他方法，如 isinstance。

逻辑运算优先级低于单独比较运算符，但采用圆括号来避免代码维护问题还是值得推荐的。

2.5.6　数值类型

Python 数值类型包括整数（int）、浮点数（float）、长整数（long）、复数（complex）。

不可修改的内置类型

这四种数值类型以及布尔类型（bool，int 的子类型），是不可变（immutable）的类型。即创建后，其值无法修改。读者可能会奇怪，明明这些变量是可以修改的。为什么说无法修改呢？请看实验：

```
>>> i = 100
>>> id(i)
35946832L
>>> i = 200
>>> id(i)
35950408L
>>> j = int(150)
>>> id(j)
35947624L
```

id 这一内置函数打印了变量所在内存的位置。所谓变量 i 被赋予新值，其实是 Python 重新初始化了一个整数类型并让 i 指向新的整数对象。但从应用开发的角度却觉得是修改了变量。而原来的整数类型对象和存储空间则被 Python 的垃圾回收机制完成回收。

实际上，这些不可修改的数值类型是使用默认构造函数构建的对象。笔者很好奇在 MicroPython 等嵌入式版本中，是否存在差异，所以也做了一模一样的实验，结论是它们的实现方式是一致的。

```
MicroPython v1.6 on 2016-04-02; NUCLEO-F401RE with STM32F401xE
Type "help()" for more information.
>>> i = 100
>>> id(i)
201
>>> i = 200
>>> id(i)
401
>>> j = int(300)
>>> id(j)
601
>>>
```

在数值类型中，Python 的一些类型范围是依赖于平台的，所以选择一个平台时需要注意这一点（尤其是涉及整数和浮点数的时候）。在 Python 2 中分为普通整数 int 和长整数。而且 Python 2 中普通整数还分 32 位和 64 位。

在 Python 2 中，32 位系统中普通整数类型精度是 32 位，范围为-2^{31}~$2^{31}-1$。而 64 位系统中的范围是：-2^{63}~$2^{63}-1$。其在不同进制中的表达方式如下。

- 十进制：默认写法，没有前后缀。
- 二进制：前缀附加 0b，如 0b11。
- 八进制：前缀附加 0，如 0123。
- 十六进制：前缀附加 0x，如 0x123。

注意

- 整数可以使用位运算，但是请记住整数是 32 位有符号整数，不是 8 位字节。
- 长整数具备无限制精度，仅受限于计算机 RAM。其表达方式采用末尾 L：999L。虽然大小写都可以，但是推荐使用大写 L。整数和长整数在 Python 3 中统一为长整数。
- 浮点类型是双精度类型，对应于 C 语言中的双精度浮点数。这一点在使用和实现嵌入式 Python 时需要摸清其浮点数是如何实现的，因为大多数嵌入式硬件如 Cortex-M4F 以及 NVIDIA GPU 都是单精度浮点数。
- 复数类型，采用大小写 J 表达虚部：12+34j，推荐使用小写 j。
- 在做两元操作时，Python 会将位数“较窄”的操作数类型转化为“较宽”的类型，即 int<long<float<complex。

2.5.7　布尔类型

和其他基本类型不同，在 Python 中，只有两个 bool（布尔）值：True/False。

```
>>> b1= True
>>> b2 = False
>>> b3 = True
>>> b4 = False
>>> id(b1)
1613110968L
>>> id(b2)
1613110552L
>>> id(b3)
1613110968L
>>> id(b4)
1613110552L
>>> type(b1)
<type 'bool'>
```

变量 b1/b3、b2/b4 的内存位置是一样的。在 Python 程序中，任何时候 bool 变量值只可以引用其中一个。结合之前的变量在内存中的表达，读者可以理解这种现象背后的原因。

2.5.8　迭代器类型

Iterator，即迭代器类型。Python 支持容器类型（container type）的迭代概念，并由__iter__和 next 方法实现。这允许用户自定义类方法来实现迭代。而各种序列类型总是支持此类方法。

所有使用 for 循环的对象类型都是可迭代对象（Iterable）。此外，还有生成器类型和 **yield** 生成器（generator）函数也归类于迭代器（Iterator）。但是，并不是所有可迭代对象都是迭代器类型。

Iterable 和 Iterator

请大家看一下代码：

```
>>> from collections import Iterator, Iterable
>>> t = tuple()
>>> l = list()
>>> m = dict()
>>> s = set()
>>> y = 'hello'
>>> isinstance(t, Iterator)
False
>>> isinstance(l, Iterator)
False
>>> isinstance(s, Iterator)
```

```
False
>>> isinstance(y, Iterator)
False
>>> isinstance((x for x in range(10)), Iterator)
True
>>> isinstance(t, Iterable)
True
>>> isinstance(l, Iterable)
True
>>> isinstance(s, Iterable)
True
>>> isinstance(y, Iterable)
True
>>> isinstance((x for x in range(10)), Iterable)
True
```

可迭代对象变成迭代器类型，需要使用 iter 函数。

```
>>> isinstance(iter(t), Iterator)
True
```

Iterator 迭代器类型是惰性计算，只有需要时才会即时计算一次。而许多容器类型虽然支持 next 方法，是可迭代 Iterable 对象，却是预先就计算完毕并存在于 RAM 中的。搞清楚这两个概念的区别可以提升性能，节省资源。

2.5.9 生成器类型

Python 生成器（generator）类型提供了一个实现迭代器的简便方式。如果容器对象__iter__方法实现为一个生成器，它将自动返回一个提供__iter__和 next 方法的迭代器对象。所以，生成器也是迭代器类型的一种。

以上这句话看上去很“学术”。我们换一种说法。列表可以通过列表推导式来创建。但受到 RAM 限制，列表容量也有限制。如果代码仅仅访问一个超大列表的一部分，那么会浪费大量 RAM 和计算资源。如果列表元素可以按照某种算法演算出来，就可以不必创建完整的列表，而节省大量的资源。这种边循环边计算的迭代机制，即生成器。

```
>>> L = [x*x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x*x for x in range(10))
>>> g
<generator object <genexpr> at 0x000000000254D3A8>
```

列表和生成器的区别在于语法中分别采用[]和()。试想，如果 range 参数不是 10 而是 10 万、1000 万，占用的资源就存在很大差别。

2.5.10　yield 表达式

在许多生成器相关设计中，会看到 yield 表达式。在计算机书籍中，yield 普遍没有直接翻译成中文。在字典中，yield 的含义非常多：产量、收益率、良率、成品率、屈服、返回等。

在 C/C++语言中没有 yield 关键字，在 Java 和 C#中是有的。在 Java 的 Thread 类的 yield 方法中，会暂停当前线程对象转而执行其他线程。Python 中的 yield 与 Java/C#类似。从字面角度来说，yield 类似于返回（return）。但与 return 语句不同的是，yield 并不“返回”到其调用者，而是暂停执行当前函数，“出让”执行权，而且内部资源并不释放。

在 Python 的异步通信框架如 Tornado/Twisted 中，因大量使用了 yield，具备了类似线程暂停的特性。但其本质上是将控制权在同一线程中的其他函数间流转，所以是协程。引入 yield 关键字的最初目的是为了让代码更加简洁，而非增加理解和调试难度。部分专注于微控制器开发的读者有 C/C++编程经验，但没有 Java/C#编程经验，yield 表达式就会如天书一般。简单易懂的 Python 开始变得高深莫测。最要命的是，传统的调试手段如 print 调试方式好像也失灵了，因为 print 的执行顺序可能因为 yield 表达式而出现变化。

yield 语句仅仅在定义生成器函数时使用，也仅用于生成器函数体内。实际上在函数定义内使用 yield 就是创建生成器函数。当生成器函数被调用时，返回一个迭代器，即生成器迭代器，也就是生成器。生成器函数体执行时是反复调用生成器的 next 方法，直到抛出异常，并停止迭代。

当 yield 语句执行时，生成器的状态被冻结，表达式列表的值会返回给 next 方法的调用者。“冻结”一次意味着所有的局部状态将会保持，包括局部变量的绑定、指令指针和内部堆栈。由于记录了足够的信息，因此在下一次 next 调用时，函数可以继续处理。

stackoverflow 上有关于 iterator、generator、yield 三者间关系的详细描述。知乎上也有类似的问答。结论是：要了解 yield 的工作原理，需要先了解生成器；而要了解生成器，需要掌握迭代器。所以，本章将按照迭代器、生成器和 yield 表达式的顺序安排内容。如果这一段内容没有读懂，读者可以按照延伸阅读内容中的实验一步步做下来体会一下。

2.5.11　序列类型

在 C 语言中，大多数原始类型是没有数据结构的。但是 Python 的许多类型本身带有或者可以扩展成复杂的数据结构，如队列、堆栈和堆，以及字典键值对等。多数 Python 程序会充分利用各种数据类型的数据结构以简化程序设计。复杂类型在 C/C++中可以使用 struct 结构体来实现；在 Java/C#中，可以使用类来实现；而 Python 可以使用容器类型的内置类型。其中，容器类型中包括序列类型和集合类型（collections）。

序列类型（Sequence）包括字符串（string）、unicode 字符串（unicode）、列表（list）、元组（tuple）、字节数组（bytearray）、缓冲区（buffer）、xrange 类型等。

字符串、unicode 字符串、元组等是不可变类型（Immutable），即一旦创建之后，存储数据无法修改，如果需要修改，则需要创建新对象保存新数据。只有列表（list）和字节数组（bytearray）是可变类型，可以就地改变对象。可变类型虽然非常灵活，但是其动态特性对性能有一定影响；而不可变类型占用内存少，执行快，而且是线程安全类型。这一点在高并发和嵌入式等性能敏感型应用中需要特别注意。

2.5.11.1 字符串

Python 的字符串类型是不可变类型。字符串是互联网和物联网数据采集中的核心数据类型。因为在计算机历史上，每个被传输的字节都必须是字符，而字符的集合就是字符串。所以在 Python 2 中，传输后的字符串要转换为整数或其他数值类型才能进一步计算。在 Python 3 中，字符串的单个成员对象是字符类型，字节串的单个成员对象是整数类型，可以直接进行数值计算。

1．创建字符串

Python 中字符串的定义是非常灵活的，可以使用单引号'，双引号"创建。如果在一行中无法完成，可以使用反斜线“\”来折叠长字符串。此外，还可用三个（单/双）引号的方式构建一个段落，包括函数的 docstring，即文档字符串也采用这种方法创建。

```
>>> a = 'abc'
>>> x = "xyz"
>>> w = '''demo string'''
>>> w
'demo string'
```

如果在字符串中包含转义符“\”，则其会被理解为转义。可使用原始字符串（raw）定义以避免使用转义符\。

```
>>> p = 'c:\some\name'
>>> print(p)
c:\some
ame
>>> p = r'c:\some\name'
>>> print(p)
c:\some\name
```

2．字符串方法

字符串的内置方法非常多。可以在 REPL 中使用 dir 方法来了解：

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattribute__',
```

```
 '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce_
_', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split', '_format
ter_parser', 'capitalize', 'center',
 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', '
isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpa
rtition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcas
e', 'title', 'translate'
, 'upper', 'zfill']
```

也可以使用 help 内置函数来了解某个特定字符串方法（内置函数），如：

```
>>>help(str.encode)
```

3．字符的容器

字符串作为序列类型的一种，是字符类型的容器。Python 2.7 的默认编码是 ASCII 英文字符。

4．unicode 字符串

unicode 是为了解决国际化和本地化而设计的文字编码。其中最常用的是可变长的 UTF-8 编码，它保证了对于 ASCII 码表的向下兼容。在 Python 2 中，字符串'abc'和 unicode 字符串 u'abc'是有区别的，必须进行转换。在 Python 3 中，字符串默认就是 unicode 字符串，没有区别。

5．字节串（bytes）

bytes 字节串需要特别说明。bytes 类型声明方式是 b'abc'。在 Python 2.X 中，字符串默认以 ASCII 形式保存，字符宽度为字节，'abc'和 b'abc'没有区别，都是 str 字符串类型，都由 char 字符类型组成。

在 Python 3.X 中，字符串默认以 unicode 形式保存，'abc'。而 b'abc'创建的类型为 bytes。

如果从物联网应用角度来看，大部分传输的内容是以英语为主的 JSON 标签或数值类型，如果要兼容 Python 2/3，则使用 b"定义字节串是相对安全的。

6．字符串的格式化

字符串的格式化采用了类似于 C 语言的%格式化符号。

```
>>> y,m,d=2016,7,8
>>> "%d-%02d-%02d"%(y,m,d)
'2016-07-08'
```

从 Python 2.6 开始，引入了 format 内置函数来实现字符串格式化，它使用花括号{}和冒号:来替代百分号%。与百分号格式化相比，它的功能更加强大而灵活。

1）通过位置格式化

```
>>> '{0},{1}'.format('kfc',20)
'kfc,20'
```

```
>>> '{},{}'.format('kfc',20)
'kfc,20'
>>> '{1},{0},{1}'.format('kfc',20)
'20,kfc,20'
```

2）通过关键字参数格式化

```
>>> '{name},{age}'.format(name='kfc',age=20)
'kfc,20'
```

3）通过对象属性格式化

```
>>> class Person:
...     def __init__(self,name,age):
...         self.name, self.age= name, age
...     def __str__(self):
...         return 'Person named {self.name} is {self.age} year old.'.format(self=sel
f)
...
>>> str(Person('kfc',20))
'Person named kfc is 20 year old.'
```

4）通过下标格式化

```
>>> person = ['kfc',20]
>>> '{0[0]},{0[1]}'.format(person)
'kfc,20'
```

5）格式限定符

format 方法中还有一个非常重要的格式限定符，用冒号:表达。

6）填充与对齐

填充常跟对齐一起使用。^、<、>分别代表居中对齐、左对齐和右对齐，其后面的参数为宽度。冒号:后面为填充的字符，只能是一个字符，默认以空格填充。

```
>>> '{:>3}'.format('123')
'123'
>>> '{:>3}'.format('123')
'123'
>>> '{:>3}'.format('1234')
'1234'
>>> '{:>6}'.format('1234')
'  1234'
>>> '{:0>6}'.format('1234')
'001234'
>>> '{:x>6}'.format('1234')
'xx1234'
```

7）精度与类型 f

浮点数精度采用类型 f 限定。

```
>>> import math
>>> math.pi
3.141592653589793
>>> '{:.5f}'.format(math.pi)
'3.14159'
```

8）其他类型

与多进制有关：b、d、o、x 分别是二进制、十进制、八进制、十六进制。主要是数值转换到各个进制的字符串类型。此外，还可以采用内置函数进行互相转换。

```
>>> '{:b}'.format(255)
'11111111'
>>> '{:d}'.format(255)
'255'
>>> '{:o}'.format(255)
'377'
>>> '{:x}'.format(255)
'ff'
```

9）千位分隔符

在财务应用中，经常使用千分位进行划分。关于这一点 format 方法也替你想到了。

```
>>> '{:,}'.format(31415926)
'31,415,926'
```

Python 2.7 和 Python 3 都支持%和 format 方法，推荐逐渐过渡到 format 方法来格式化字符串。

2.5.11.2　列表

Python 中最常用的数据结构是列表。首先，它是一种容器类型，可以包容其他类型。其次，它可以与 C 语言中的数组类比。最后，也可以将其作为队列或者堆栈一样使用。相对应地，其占用的资源也较多。所以，在物联网报文处理、文件处理等场景中需要注意避免使用列表占用大量资源的情况。

Python 可以使用方括号[]创建列表。

```
>>> la = [0,1,2,3,4,5,6]
>>> la
[0, 1, 2, 3, 4, 5, 6]
>>> el = []
>>> len(el)
0
>>> sl = [1]
>>> sl
```

```
[1]
>>> sl = [1,]
>>> sl
[1]
>>> len(sl)
1
```

在创建单一条目的列表中，逗号不是必需的。此外，只要是序列类型如字符串、元组，都可以用来创建列表。

```
>>> l = list('Hello World')
>>> l
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r',  'l', 'd']
>>> t = 1,2,3,4,5
>>> type(t)
<type 'tuple'>
>>> l = list(t)
>>> l
[1, 2, 3, 4, 5]
```

1．访问列表条目

访问列表同样可以采用方括号[]下标进行访问。也可以采用 list[start:end:step]进行切片。其中 start 和 end 为开始和结束位置索引，step 是切片要跨过的条目数量。此外，还可以在结束索引中使用负值，即从结尾倒数。也可以忽略结束值。

```
>>> l = list([1,2,3,4,5,6,7,8,9,10])
>>> l[2]
3
>>> l[1:5]
[2, 3, 4, 5]
>>> l[1:-2]
[2, 3, 4, 5, 6, 7, 8]
>>> l[1:10:3]
[2, 5, 8]
>>> l[1::2]
[2, 4, 6, 8, 10]
>>>
```

切片是容器类型的强大方法：列表还支持倒序切片、只有步进值切片，以及没有参数直接赋值，功能非常强大。

MCU 资源（RAM）有限，无法针对单一物理量做长时间缓存。最常见的做法就是实现循环数据采集，若干 ADC 的数值对应若干物理量，在 RAM 环形缓冲区中缓存，然后定时上传给服务器处理。轮询 ADC 的结果就是所有物理量基于时间戳上传，并隐性地进行了数据复用。在物联网实际工程中，采用各个物理量附加数据采集时间戳的形式非常常见，如：

时间戳：温度|湿度|压强|风速|流量……

但是在应用端，则需要针对单一物理量如温度进行绘图。所以，这提出了数据解复用的需求。而列表切片就是最简单的数据解复用方法。因为步进数值是固定的，所以可以按照固定偏移量快速地进行切片转换。

```
sample = b"\x01\x02\x03\04\x05" # sensor data sample
datastream = sample * 3
temperature = datastream[0::5]
humidity = datastream[1::5]
pressure = datastream[2::5]
windspeed = datastream[3::5]
flow = datastream[4::5]
```

2．修改列表条目

列表是一个可变的序列，这意味着不但可以访问列表条目，还可以修改它们。如果要添加条目，可以使用 append 方法。

```
>>> l = list([1,2,3,4,5,6,7,8,9,10])
>>> l[3]=11
>>> l
[1, 2, 3, 11, 5, 6, 7, 8, 9, 10]
>>> l.append(100)
>>> l
[1, 2, 3, 11, 5, 6, 7, 8, 9, 10, 100]
```

3．异构的可变序列

作为容器类型，列表可以包含其他类型，甚至不同类型的数据和对象。所以列表可以成为复杂数据结构的基础，比如数组、堆、堆栈和各种复合类型。

4．数组

如果列表中包含另一个列表，在此基础上可以拓展出二维数组和三维数组。

```
>>> al = [[0,1,2],[3,4,5],[6,7,8]]
>>> al[0][0]
0
>>> al[2][2]
8
>>> al = [[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
>>> al[0][0][1]
1
>>> len(al)
2
>>> len(al[0])
2
>>> len(al[0][0])
2
```

5．作为堆栈或队列

列表有许多非常便利的方法，比如排序和反转等。这些操作都会直接操作所在的列表。除可以作为数组使用以外，还可以构成队列（即 FIFO，先进先出）以及堆栈（即 LIFO，后进先出）的数据结构。所谓“进”是通过 append 方法实现的，“出”则由 pop 方法来实现。堆栈结构列表可以采用 pop 弹出最后一个条目；如果弹出的是第一个条目，采用 pop(0)。

```
>>> l = list('abcdefghijk')
>>> id(l)
41147464L
>>> l.reverse()
>>> l
['k', 'j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
>>> id(l)
41147464L
>>> l.sort()
>>> l
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
>>> id(l)
41147464L
>>> l.pop()
'k'
>>> l
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>> l.pop(0)
'a'
>>> l
['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>>
```

虽然列表可以构成堆栈，但是效率却不高。所以 Python 特别实现了 collections.deque 类，速度较快：

```
>>> from collections import deque
>>> queue = deque(["NY","LA","SF","NJ","OH"])
>>> dir(queue)
['__class__', '__copy__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__init__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__redu
ce__', '__reduce_ex__', '__repr__', '__reversed__', '__setattr__', '__setitem__', '_
_sizeof__', '__str__', '__subclasshook__', 'append', 'appendleft', 'clear', 'count',
 'extend', 'extendleft', 'maxlen', 'pop', 'popleft', 'remove', 'reverse', 'rotate']
>>> queue.append("VC")
>>> queue.append("TX")
>>> queue
deque(['NY', 'LA', 'SF', 'NJ', 'OH', 'VC', 'TX'])
>>> queue.popleft()
```

```
'NY'
>>> queue.pop()
'TX'
>>> queue
deque(['LA', 'SF', 'NJ', 'OH', 'VC'])
```

读者有空研读一下 collections 的用法或许可以找到更合适的数据结构和类型。

6．列表推导式

列表推导式（List Comprehension），也被翻译为列表解析式、列表生成式。该方式是创建列表的简单方法，主要通过迭代方法按照某种计算规则建立列表。通常，我们是通过迭代语句来生成列表的：

```
>>> sq = []
>>> for x in range(10):
...     sq.append(x**2)
...
>>> sq
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

采用列表推导式方法：

```
>>> sq = [x**2 for x in range(10)]
>>> sq
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

下面是更复杂些的推导式：

```
>>> from math import pi
>>> [round(pi,i) for i in range(1,6)]
[3.1, 3.14, 3.142, 3.1416, 3.14159]
```

上例中已经采用内置函数 round；也可以采用匿名函数或自定义函数。

嵌套数组的推导方式如下：

```
>>> [(x,y) for x in [1,2,3] for y in [8,2,0]]
[(1, 8), (1, 2), (1, 0), (2, 8), (2, 2), (2, 0), (3, 8), (3, 2), (3, 0)]
>>> [(x,y) for x in [1,2,3] for y in [8,2,0] if x!=y]
[(1, 8), (1, 2), (1, 0), (2, 8), (2, 0), (3, 8), (3, 2), (3, 0)]
```

数组维度的转换如下：

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
>>> [[row[i] for row in matrix] for i in range(4)]
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
>>> zip(*matrix)
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

虽然采用列表推导式很容易进行维度转换，但是使用内置函数 zip 则更加容易。zip 指的是将序列类型压缩返回列表，包含一维的 tuple 元组。

数组降维：

```
>>> [i for row in matrix for i in row]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>>
```

关于列表在文本操纵、CSV 文件读取等方面的更多例子，读者可阅读本章延伸阅读内容。

7．del 语句

del 语句可以用来删除对象，以及对象中的部分元素。del 语句允许根据索引来删除元素。与 pop 只能够弹出一个不同，del 语句还可以用于从列表中删除切片，甚至删除整个列表。

```
>>> al = [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del al[0]
>>> al
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del al[2:4]
>>> al
[-1, 0, 3, 4, 5, 6, 7, 8, 9]
>>> del al[:]
>>> al
[]
>>> del al
>>> al
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'al' is not defined
```

因为 al 已经被删除，所以最后引用名称 al 会出错。

2.5.11.3 元组

前面我们看到字符串和列表具备许多共同的属性和方法，如索引和切片。它们同是序列类（str/unicode/list/tuple/bytearray/buffer/xrange）中的成员。和列表最类似的标准序列类型是元组。

元组和列表虽然很类似，但它们经常用于不同场景和目的。列表是可变的数据类型。其元素可以通过索引和迭代访问，并就地修改。所以，列表具备更多方法。元组是不可改变的类型，所以没有 append 之类的方法。但采用元组，可以节省资源。元组可以使用圆括号和逗号来定义。

作为容器类型之一，Python 中的元组类型可以存放不同的对象。为变量分配以逗号分隔的对象序列，就可以创建元组。

```
>>> t = (0,1,2,3,4)
>>> type(t)
<type 'tuple'>
>>> x = 'a','b','c'
>>> type(x)
<type 'tuple'>
>>> y = tuple()
>>> type(y)
<type 'tuple'>
>>> z = (3,)
>>> type(z)
<type 'tuple'>
```

元组的创建需要圆括号和逗号。在大多数情况下，没有圆括号()也可以创建元组。在某些上下文中，圆括号是必需的。但在创建单一条目的元组中，逗号是必需的。所以，使用元组类型需要留意元素数量为 0 或 1 个的元组。

```
>>> empty_tuple = ()
>>> len(empty_tuple)
0
>>> singleton = "hello",
>>> len(singleton)
1
>>> singleton
('hello',)
>>> single = "hello"
>>> len(single)
5
>>> type(singleton)
<type 'tuple'>
>>> type(single)
<type 'str'>
```

上例中 singleton 赋值末尾的逗号，将其类型声明成元组，而非普通字符串。即便没有使用圆括号，只要最后以逗号结尾就是元组。如果需要声明一个元素的元组，那么尾部的逗号也是不可少的，否则就是字符串类型。

1．访问元组

可以使用方括号[]来访问元组。Python 采用零排序，即第一个对象下标为零。而访问数据项的索引需要声明越过的数据项（序列）距离。如获得第三个数据项，则需要从（包含）第一个数据项起越过两个数据项。距离就是 2。还可以使用分段方法来提取并创建新的元组。

```
>>> t = (0,1,2,3,4,5)
>>> t[3]
3
>>> t[1:3]
```

```
(1, 2)
>>> t[1:5:2]
(1, 3)
```

2．异构元组

```
>>> x = (0,'one',2.0,'three',4,(5,6,7))
>>> x[1:4]
('one', 2.0, 'three')
>>> type(x[5])
<type 'tuple'>
>>> type(x[3])
<type 'str'>
>>> t[4]='four'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

在此实例中，一个元组中可以包含另外一个元组，形成异构元组。最后一个 TypeError 表明：元组是不可变的。其值无法修改。

3．元组合并

```
>>> x = (0,'one',2.0,'three',4,(5,6,7))
>>> x[5][2]
7
>>> x1 = (100,101,102)
>>> y = x1 + x
>>> y
(100, 101, 102, 0, 'one', 2.0, 'three', 4, (5, 6, 7))
```

元组可以合并成新的元组，并可以递归访问。

此外，元组虽然是不可改变类型，但它包含的类型却可以是可变类型。

```
>>> x = range(10)
>>> type(x)
<type 'list'>
>>> y = x,
>>> type(y)
<type 'tuple'>
>>> y[0].append(100)
>>> repr(y)
'([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 100],)'
```

4．元组打包和解包

```
>>> a = 100
>>> b = 200
>>> c = 300
```

```
>>> t = (a,b,c)
>>> t
(100, 200, 300)
>>> x,y,z = t
>>> x
100
>>> y
200
>>> z
300
```

和 C 相比，元组打包和解包都很容易。在 Python 中，经常看到函数可以返回多个数值，分别赋值给变量。这是因为 Python 函数返回值类型事实上就是元组类型。

2.5.11.4　字节数组

字节数组是可变类型，采用 bytearray 内置函数构造。在 REPL 中，输入 help(bytearray)可以获得相关信息。字节数组的来源可以是：

- 可迭代的整数序列，整数范围为 0~255；
- 字符串；
- 字节或者另外的字节数组对象；
- 任意实现了缓冲区 API 的对象。

```
>>> x = bytearray('\x12\x34\x56\x78')
>>> x
bytearray(b'\x124Vx')
>>> x = bytearray('abcdef')
>>> x
bytearray(b'abcdef')
>>> x = bytearray(list('abcdef'))
>>> x
bytearray(b'abcdef')
```

字节数组出现在 Python 2.6 之后的版本中。字节数组属于可变类型，与数组类型不同的是，字节数组的每个元素必须是 byte 字节。字节数组常用于通信中，其字节元素可以作为整数参与计算。

2.5.11.5　数组（array）

与内置类型 list 和 bytearray 不一样，array 是作为标准库的一部分而存在的。由于不是内置类型，所以必须先导入 array 模块。array 模块定义的是一种序列数据结构，和 list 很类似，但是其所有成员必须是相同类型。array 是高效管理固定类型数据的序列类型，所以有必要单独展开介绍一下。

构建函数方法： class array.array(typecode[,initializer])，可以从 string、unicode、list 这些类型上初始化。

```
>>> import array
>>> buf = array.array('i',[0 for i in range(10)])
>>> buf
array('i', [0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> buf = array.array('c','hello world')
>>> buf
array('c', 'hello world')
```

由于 C/Python 类型存在不同，在转换成数组时，需要注意其字节数，具体长度如表 2-3 所列。

表 2-3 C 与 Python 类型字节数比较表

类型编码	C 基本类型	Python 类型	字节数
'c'	char	char	1
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	long	2
'l'	signed long	int	4
'L'	unsigned long	long	4
'f'	float	float	4
'd'	double	float	8

2.5.11.6 缓冲区对象

缓冲区对象将内存的一个连续区域模拟为单字节字符序列。Python 没有直接构建缓冲区对象的语句，而是使用内置函数构建：

```
buffer(obj[,offset[,size]])
```

缓冲区对象与对象共享内存，对于字符串切片和其他基于字节数据操作，效率非常高。另外，缓冲区对象还可以用来访问其他 Python 类型存储的原始数据，如数组、unicode 字符串。缓冲区对象是否可变，取决于对象本身。

例如：

```
>>> s = "Hello world!"
```

```
>>> id(s)
41382512L
>>> t = buffer(s, 6, 5)
>>> t
<read-only buffer for 0x0000000002777270, size 5, offset 6 at 0x0000000002837FB8>
>>> print t
world
>>> type(t)
<type 'buffer'>
>>> s = bytearray(1000)
>>> t = buffer(s, 1)
>>> s[1] = 5
>>> t[0]
'\x05'
```

在第一个字符串例子中，从字符串 s 中，下标位置 6 开始截取 5 个字节，构成缓冲区 t。整个过程中并没有产生新的字符串，而是通过引用来截取字符串。

在第二个字节数组例子中，从字节数组中截取第 1 个字节。由于字节数组是可以修改的，所以可以通过索引来修改。

总的来说，缓冲区类型可以帮助优化存储器的使用。在 Python 3 中，buffer 被命名更加确切的 memoryview 所取代；在 Python 2.7 中，两者都可以使用。

2.5.11.7　xrange 类型

xrange 类型是一种常见于循环的不可变序列。其优点是 xrange 对象总是返回同样类型的存储器，而与其代表的范围无关。xrange 对象的行为方法很少，仅支持索引、迭代和长度方法。不支持切片、连接和重复，并且在该类型上实施 in、not in、max、min 效率较低。

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(range(10))
<type 'list'>
>>> xrange(10)
xrange(10)
>>> type(xrange(10))
<type 'xrange'>
```

可以看到，我们代码常用的循环控制：

```
for i in range(x):
```

其实是被展开为一个列表了：

```
for i in [0,1....x]:
```

设想如果 x 是一个很大的整数，这会占用多少内存？尤其 Python 的整数类型所占的内存可

不止是 8/16/32 位，而且列表是比较耗费资源的容器类型。即使是强大的服务器，采用 xrange 惰性计算的 xrange 替代 range 也可以节省内存，提高性能。

在 Python 2 中，range/xrange 是两种实现。但在 Python 3 中，移除了 range 实现，并将 xrange 改名为 range。即，Python 3 中的 range 是 Python 2 中的 xrange，但增加了一些新的属性和方法。

2.5.12 set 集合类型

为了区别两种集合类型，防止混淆，下面会注明 set 和 collections 原文。

set 集合类型指的是可以哈希（hash）计算的、无序的、无重复的数据集合，包括 set、frozenset。常见的使用方法包括成员测试，从序列中删除重复项和数学计算（交集、并集、差集和对称差）。这种数据结构可以用于一些有趣的应用中，如互为好友的计算等。其中，frozenset 是不可改变类型。frozenset 无法修改，所以 set 所具备的 add/pop/discard/remove 和所有*_update 方法都不支持。

可以采用花括号{}或 set()内置函数构建 set 集合类型。如果要构建空的 set，则必须使用 set()。

```
>>> set()
set([])
>>> {1,2,3,4}
set([1,2,3,4])
>>> sa = set(range(10))
>>> sc = set(range(5,15))
>>> sa
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> sc
set([5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
>>> sa & sc
set([8, 9, 5, 6, 7])
>>> sa.intersection(sc)
set([8, 9, 5, 6, 7])
>>> sc.intersection(sa)
set([8, 9, 5, 6, 7])
>>> sa|sc
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
>>> sa.union(sa)
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> sa.union(sc)
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
>>> sc.union(sa)
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
>>> sa.update(sc)
>>> sa
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
>>> sa = set(range(10))
```

```
>>> sa
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> sa-sc
set([0, 1, 2, 3, 4])
>>> sa.difference(sc)
set([0, 1, 2, 3, 4])
>>> sa.difference_update(sc)
>>> sa
set([0, 1, 2, 3, 4])
>>> sa = set(range(10))
>>> sa^sc
set([0, 1, 2, 3, 4, 10, 11, 12, 13, 14])
>>> (sa-sc)|(sc-sa)
set([0, 1, 2, 3, 4, 10, 11, 12, 13, 14])
>>> sa.symmetric_difference(sc)
set([0, 1, 2, 3, 4, 10, 11, 12, 13, 14])
>>> sc.symmetric_difference(sa)
set([0, 1, 2, 3, 4, 10, 11, 12, 13, 14])
>>> sa.symmetric_difference_update(sc)
>>> sa
set([0, 1, 2, 3, 4, 10, 11, 12, 13, 14])
```

上面提到过列表推导式，set 集合类型也有 set comprehension：

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```

2.5.13　映射类型

一个映射对象将值映射到任意对象。映射是可变对象。其主要是字典（dictionary）类型，以及衍生类型，如 collections.defaultdict。

字典类型中的键可以是任意值，但不可以是无法哈希的值，如列表、字典和其他可变类型的值不可以用作键。因为这些类型通过值而不是对象 ID 进行比较。字典通过在花括号{}中放置逗号分隔的 key:value 对进行定义，或通过 dict 构造函数创建。

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one':1, 'two':2, 'three':3}
>>> c = dict(zip(['one','two','three'],[1,2,3]))
>>> d = dict([('one',1),('two',2),('three',3)])
>>> e = dict({'two':2, 'three':3, 'one':1})
>>> a == b == c == d == e
True
>>> id(a), id(b), id(c), id(d), id(e)
(41166504L, 41166232L, 41166776L, 41167048L, 41167320L)
```

从代码中可以看到 a、b、c、d、e 的值一样，但是 id 是不一样的。

采用内置函数 keys 可以得到一个字典的所有键的列表。

```
>>> a = dict(a=1,b=2,c=3)
>>> a
{'a': 1, 'c': 3, 'b': 2}
>>> a.keys()
['a', 'c', 'b']
```

字典类型也可以通过键值对序列、字典推导式（dict comprehension）以及更加常见的关键字参数进行赋值。

```
>>> dict([('allan',1),('andrew',2),('armanda',3)])
{'armanda': 3, 'allan': 1, 'andrew': 2}
>>> {x:x**2 for x in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
>>> {str(x):x**2 for x in range(5)}
{'1': 1, '0': 0, '3': 9, '2': 4, '4': 16}
>>> dict(chinese=100,english=99,math=90)
{'math': 90, 'chinese': 100, 'english': 99}
```

此例已经告诉读者，Python 的关键字输入参数实际上就是一个字典。

在互联网和物联网应用中，键值类型大量用于 HTTP 请求、存储等方面：各类 NoSQL 本质上就是各种键值类型数据的演化版本；JSON 本质上也是键值类型的应用。当这些类型数据被导入 Python，成为 Python 的字典类型对象时，可以很容易地进行处理。

2.5.14 其他类型

所谓其他类型都是针对特定目的的类型，其使用方法各异。

2.5.14.1 文件对象

文件对象由 C Stdio 模块构成，可以采用内置函数 open 创建。在 Windows 中，可以简单地将其理解为与操作系统和文件系统的接口，是系统配置和数据持久保持的媒介。但在 Linux 系统中，设备也是文件。Linux 的文件对象和物联网的各类模块与通信的关联远比 Windows 要重要得多。在 5.3 节中会从 Linux 的设备文件开始介绍，Python 应用程序都是从一些系统调用，如 read/write/fctrl/ioctrl 访问这些设备的。

2.5.14.2 memoryview 类型

内存视图（memoryview）允许 Python 无须复制，即可以访问支持缓冲区协议的对象内部数据。存储器常常被解释为字节。对于简单类型如 str 和 bytearray，可以返回字节，但是其他类型会返回较大的数据。前面提到，memoryview 其实是 buffer 类型，这种类型在调试中或许可以

帮上大忙。

2.5.14.3　上下文管理器类型

Python 的 with 语句支持上下文管理器定义的运行时上下文概念。分别通过定义 enter/exit 两个方法，允许用户类定义运行时上下文。这允许把常见的 try...except...finally 模式封装起来重用。

Python 的生成器和 contextlib.contextmanager 装饰器提供了一个实现这些协议的简单方法。通过 with 语句可以简化异常捕获的代码。with 语句的语法如下：

```
with context_expression [as target(s)]:
    with-body
```

target 可以不止一个。

在文件访问中，打开文件使用 try...except...finally 是比较常见的。

```
logfile = open(r'twistd.log')
try:
    for line in logfile:
        print line
except:
    pass
finally:
    logfile.close()
```

使用上下文管理器可以使得代码更加简单。

```
with open(r'twistd.log') as logfile:
    for line in logfile:
        print line
```

上下文管理器类型支持的还有线程 Threading 和 Decimal 等模块。此外，用户也可以自定义支持上下文管理器协议的类。

2.5.14.4　模块

对于模块类型，Python 仅支持对于模块名称属性的访问。

```
>>> import os
>>> type(os)
<type 'module'>
```

2.5.14.5　类和类实例

```
>>> class C(object):
...     def __init__(self):
...         pass
...
>>> c = C()
```

```
>>> a = C()
>>> type(a)
<class '__main__.C'>
>>> type(a.__init__)
<type 'instancemethod'>
```

2.5.14.6 函数

函数对象通过函数定义关键字 def 创建，访问方式为 func_name（func_args）；包括 Python 内置函数和用户自定义函数。

```
>>> def f(x):
...     x = x+2
...
>>> type(f)
<type 'function'>
```

不同平台、不同版本的 Python 在内置函数上有很大差异，尤其是嵌入式 Python 尤其如此，其往往只实现部分子集。

2.5.14.7 方法

方法是使用属性符号调用的函数，包括内置方法和类的实例方法。

```
>>> type(abs)
<type 'builtin_function_or_method'>
```

2.5.14.8 代码对象

代码对象用于“编译”的可执行 Python 字节码。代码对象由内置 compile 函数返回，并从函数对象的 func_code 属性中提取。

2.5.14.9 类型对象

类型对象用于不同对象的类型，通过内置函数 type 访问。

```
>>> x = u'Hello'
>>> type(x)
<type 'unicode'>
>>> type(type)
<type 'type'>
```

2.5.14.10 空对象

空对象或者空值是一个特殊的值，用 None 表示。None 与 0 不同，0 为整数值，而 None 代表空。

```
>>> type(None)
<type 'NoneType'>
```

2.5.14.11　省略号

Ellipsis 类型，此对象用于扩展的切片表示法，以省略号表达。

```
>>> type(Ellipsis)
<type 'ellipsis'>
```

2.5.14.12　NotImplemented 对象

在不同版本的 Python 中，某些方法未被实现，或者某种类型的操作并没有实现，都会抛出此类对象。比如笔者尝试在 MicroPython 中使用 str 切片操作，系统就抛出了 NotImplemented 异常。

```
>>> type(NotImplemented)
<type 'NotImplementedType'>
```

2.5.14.13　内部对象

包括 Python 的堆栈帧对象、切片对象等，与 Python 内部实现有关联。

2.5.14.14　内置异常

异常也是类的对象，定义在 exceptions 模块中，是 exception.BaseException 的对象。

2.5.15　控制流

Python 的控制流语句与大部分语言如 C/C++很类似，但稍有区别。

2.5.15.1　if...elif…else

if...elif…else 是常见控制流，其中 if 是必需项，else 是可选项，elif 是 else if 的缩写方式。C/Java 语言中常见的 switch...case 语句在 Python 语法中以 if...elif 来替代。

```
if x<0:
    print "-"
elif x==0:
    print "0"
elif x>0:
    print "+"
```

但其实 Python 的完整控制流是 if...elif...else...。但 else 是可选项，可以省略。

2.5.15.2　for 语句

Python 的 for 语句和其他语言如 C/Java 有所不同，它按照序列元素出现的顺序进行迭代。

```
>>> ws = ['Hello', 'world,', 'I', 'am', 'Python']
>>> for s in ws:
...     print s.upper()
```

```
...
HELLO
WORLD,
I
AM
PYTHON
>>>
```

2.5.15.3 range 函数

在 Python 2.7 中，range 可以用来生成等差列表；在 Python 3 中，range 是返回序列类型。

2.5.15.4 break/continue/else 语句

break 语句与 C 语言类似，用于跳出最里层的 for/while 循环。但和 C 语言不同的是：for/while 循环可以有 else 子句。else 子句与 for/while 的缩进位置一致，当 for 循环条件结束或 while 判断条件为 False 时，立即执行 else 子句中的语句。除非在 for/while 循环中采用 break 语句跳出循环，此时 else 子句也不再执行。

```
>>> for n in range(2,10):
...     for x in range(2,n):
...         if n%x == 0:
...             print("%d = %d * %d"%(n,x,n/x))
...             break
...     else:
...         print("%d is a prime"%(n))
...
2 is a prime
3 is a prime
4 = 2 * 2
5 is a prime
6 = 2 * 3
7 is a prime
8 = 2 * 4
9 = 3 * 3
```

continue 语句与 C 语言中使用的方式类似，表示继续下一次迭代。

2.5.15.5 pass 语句

pass 语句什么也不做（在语法上必须要有一条语句占据某些位置）。在编写新代码时其常用于临时保留位置，创建最小的类、方法、函数。

```
class EmptyClass:
    pass

def doNothing():
     pass
```

2.5.15.6　return 语句

return 返回只发生在嵌套的函数定义中，而非嵌套的类定义中。return 默认返回 None，或将表达式作为返回值。当 return 将控制权从带有 finally 子句的 try 语句带回时，finally 子句的操作会在实际离开函数前执行。

在生成器（generator）函数中，return 语句不允许包含表达式。这意味着生成器已经结束，会引发 StopIteration 异常。

```
>>> def myfunc():
...     pass
...
>>> x = myfunc()
>>> x
>>> type(x)
<type 'NoneType'>
```

注意　没有返回值的 return 语句，甚至没有 return 语句，都等价于 return None。

2.5.15.7　yield 语句

当然，上面提到的 yield 语句也是控制流的一种。其用于将控制权流转给当前线程的其他函数和方法。

2.5.16　内置函数

读者可以在 REPL 下使用 dir(__builtins__)命令来探索一下，表 2-4 列出了笔者系统内 Python 2.7 的内置函数。与物联网直接相关的内置函数大多涉及各个基本数据类型之间的转换，而且 Python 2.7 和 Python 3.X 在某些实现细节上不一样。

表 2-4　内置函数

abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()

续表

chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

2.5.17 用户自定义函数

Python 采用关键字 def 来定义函数，后跟函数名和圆括号()标明的形参列表。函数体从第二行开始必须缩进。

```
>>> def myfunc(x,y):
...     '''a demo function'''
...     print(x+y)
...
>>> myfunc(10,1)
11
>>> myfunc.__doc__
'a demo function'
```

函数体的第二行可以是所谓的 docstring，即文档字符串。这主要用于对函数的说明性文字，并可以使用__doc__来引用。

执行函数会引入一个该函数的局部变量符号表。函数中所有的赋值操作都将值存储在局部符号表中。变量引用的顺序是，首先查找局部符号表，然后是上层函数局部符号表，接着是全局符号表，最后是内置名字表。所以，在函数体内可以引用全局变量，但不可以对全局变量直接赋值，必须采用 global 关键字声明全局变量后，才可以赋值。

```
>>> pi = 3.1415
>>> def df(x):
...     print(2*pi)
...
>>> df(3)
6.283
```

即使没有 global 声明，在函数内也可以引用全局变量。

```
>>> def df(x):
...     print(2*pi)
```

```
...     pi = 3
...
>>> df(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in df
UnboundLocalError: local variable 'pi' referenced before assignment
```

如果没有 global 声明，则在函数内无法修改全局变量。

```
>>> def df(x):
...     global pi
...     print(2*pi)
...     pi = 3
...
>>> df(3)
6.283
>>> pi
3
```

函数调用的实参在函数调用时引入被调用函数的局部符号表。参数传递采用传值调用，即对象的引用。函数定义会在当前符号表引入函数名。函数名对应值的类型是自定义函数。函数名可以分配给另一个变量，也可以作为函数调用。这有点儿像函数别名。这也是许多高级技巧如装饰器等技巧的实现基础。

还是以前面的 df 函数为例。

```
>>> df
<function df at 0x0000000002773CF8>
>>> demo = df
>>> demo(3)
6
```

函数体没有定义函数返回值时，Python 函数默认返回 None 类型。此外，Python 可以返回多个对象，这些对象被封装在一个元组中返回。最后，函数返回值也可以是另一个函数。

2.5.17.1　函数的参数

Python 函数的参数类型有四种：

- 必选参数，又称位置参数，required argument/positional argument；
- 默认参数，default argument；
- 可变参数，又称任意参数，variable length argument/arbitrary argument；
- 关键字参数，keyword argument。

参数的组合顺序是：必选参数、默认参数、可变参数、关键字参数。

2.5.17.2 必选参数

```
def myfunc(x,y):
    return x*x+y

>>> def myfunc(x,y):
...     return x*x + y
...
>>> myfunc(3,10)
19
```

myfunct 的参数 x、y 都是必选的位置参数，即参数必须按照参数位置一一对应传入。

2.5.17.3 默认参数

在许多情况下，一些参数在函数定义时可以设定默认值，以减少函数调用时的冗余输入信息。无论是输入部分必需的参数，还是输入全部参数，函数定义只需要一个。作为对比，在其他语言中，C++支持默认参数，而 Java 则通过重载来实现。

```
def user(name, password, age=None):
    if age:
        mysql_insert(name, password, age)
    else:
        mysql_insert(name, password)
```

默认参数可以用于修改比较少、重复度较高的参数，即大多数共享一个默认值，或者作为选项参数（即默认值为 None）提供。在设定默认参数时，默认参数必须是不可变的对象。如果是可变类型，如列表，则计算结果会传递给下一次函数调用。要避免此类问题，则可以使用 None 替代。

2.5.17.4 可变参数

Python 中常见的可变参数是*args，其含义是让函数接收任意数量的参数。这些参数放置在元组中传递给函数。

```
def write_multiple_items(file, separator, *args):
    file.write(separaotor.join(args))
```

实际上在函数装饰器中会用到此类可变参数，在命令行输入中也会用到可变参数。这种情况的特点是无法预知输入参数的数量和类型。

2.5.17.5 关键字参数

Python 中常见的关键字参数是**kw，参数接收的是一个字典。其原始形式如下：

```
def shield(voltage, input='digital', output='dac', type='arduino'):
    pass
```

```
def bus(voltage, **io):
    pass
```

当这些默认参数的数量不确定时，可以使用**kw 来传递一个字典类型参数。通过这种方式，我们可以很容易地扩展函数而不需要重新定义函数。而且，采用关键字参数的另外一个好处是，参数传递不必遵循传递顺序。关键字参数与默认参数的区别在于：关键字参数用于函数调用，而默认参数用于函数定义。

2.5.17.6　递归函数

一个函数如果可以调用自己，就是递归函数。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。虽然如此，但使用递归函数需要注意防止栈溢出（stack overflow）。这是因为函数调用在计算机中是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧。由于计算机存储器是有限的，所以堆栈区也是有限制的，一个无法收敛的递归函数直接危害系统安全。

2.5.18　模块

模块是包含 Python 定义和声明的文件。文件名是模块名称加上.py 文件后缀。

在 REPL 交互环境中只能够运行小型代码，大型程序需要先编写在文本中，然后运行。随着程序越来越长，许多代码会被反复调用，可以将程序分解为多个文件，进行模块化设计。Python 将这种用户自定义文件称为模块，模块中的定义可以导入其他模块。

可以在某个路径下编写 demomod.py，内容如下：

```
#demomod.py

def func(n):
     print(n**2)

def func2(n):
    print(n**3)
```

进入 REPL 后输入：

```
>>> import demomod
>>> demomod.func(3)
9
>>> demomod.func2(3)
27
>>> f = demomod.func
>>> f(3)
9
```

最后是将模块中的函数名赋值给本地变量，并调用该函数。

导入模块可以采用以下格式：

```
import modulename
from modulename import func1, func2
from modulename import *
```

不同的导入方式，引用函数的方式是不一样的，这主要体现在函数的前缀上。其中，采用import *会一次性将模块或者包中的所有函数或方法导入。

2.5.18.1 模块作为脚本执行

模块也可以是脚本。如果要将某个模块作为脚本运行，则需要在最后增加一些判断：

```
if __name__=='__main__':
    func(3)
```

在命令行中可以使用 pythonmodulename.py 形式运行，或者直接运行 modulename.py 也可。

2.5.18.2 模块搜索次序和路径

当导入某个模块如 mod 时，解释器首先搜索名为 mod 的内置模块。如果没有找到，则会尝试在 sys.path 变量的路径中查找 mod.py 文件。sys.path 的初始值来自：

- 脚本当前目录；
- PYTHONPATH；
- 与安装相关的默认值。

脚本当前目录是搜索的第一顺序，所以当前目录中如果有与标准库同名的模块，将被优先加载，而同名标准库将不会被加载。

```
>>> import sys
>>> sys.path
['', 'C:\\Python27\\lib\\site-packages\\pythondoc-2.1b7.post20070909-py2.7.egg', 'C:
\\Windows\\system32\\python27.zip',
'C:\\Python27\\DLLs', 'C:\\Python27\\lib', 'C:\\Python27\\lib\\plat-win', 'C:\\Pyth
on27\\lib\\lib-tk', 'C:\\Python27', '
C:\\Python27\\lib\\site-packages', 'C:\\Python27\\lib\\site-packages\\wx-3.0-msw']
```

增加路径的方式一是在 sys.path 列表中添加自己的路径。

```
sys.path.append('/usr/allankliu/my_python_module')
```

方式二是设置环境变量 PYTHONPATH。

2.5.18.3 编译过的 Python 文件

执行“编译”过的 Python 程序与执行 Python 源码相比，启动速度要快。这些文件采用同

样的模块名称，并采用.pyc 做后缀。此外，还有一种同名的.pyo 文件，是一种“优化”过的 pyc 文件。在 pyc 文件中将记录原始 py 文件的时间戳，如果时间不匹配，.pyc 将被 Python 主程序忽略，或重新编译。

由于 python bytecode 是跨平台的，所以可以作为交付给用户的代码。这也是一种“二进制”交付文件的方式，一定程度上可以帮助用户实现商业化软件。

一般来说，运行 Python 源码后，Python 解释器会将所有模块从.py 源码编译成.pyc 字节码文件。但有时候手动替换已经安装后的模块，可能不再产生 pyc 字节码文件。需要采用以下命令直接编译。

```
$ sudo python -m compileall redis.py
```

在以上例子中，redis.py 就是需要编译的模块。

不同 Python 的字节码可能是不同的，所以需要对应版本的 Python 进行编译。

2.5.18.4　反编译

若读者需要将 pyc 反编译成 py 文件，这方面的信息请在互联网中检索 uncompyle2、decompyle2、DePython、unpyc、uncompyle、pycdc 等模块。对 Python 2.7 而言，uncompyler 是最好用的工具。

2.5.18.5　标准模块

Python 的标准模块非常丰富，可以参考 2.6 节简单了解大致的标准模块分类。关于详细使用说明，读者可以查看 Python 安装后自带的文档，或查看官方网站文档。

2.5.18.6　安装第三方模块

安装第三方模块有 easy_install、pip、源码安装等方式。其中，easy_install 属于较早的包管理工具，现在比较推荐 pip。但有些第三方模块（尤其是含有扩展的模块），有时候需要从源码编译安装或者使用更加特殊的方式进行安装。如果无法利用 pip 安装，请先查阅其官网的安装说明。

假设我们从 PyPI 官网安装 pyserial，可以使用以下命令：

```
# pip install pyserial
```

2.5.18.7　dir 函数

前面我们已经使用 dir 函数来找出模块中定义的名字。它将返回一个排序后的字符串列表，包括变量、模块和函数。

- 可以使用 dir(modname)来了解特定模块的名字。
- 还可以使用不含参数的 dir 来了解当前已定义名称。

- 关于内置函数和变量，可以使用 dir(__builtins__)来了解。

2.5.18.8 包（Package）

包是一种管理 Python 模块命名空间的方式，采用小数点风格。模块 a.b 代表包 a 中有一个 b 模块。这种方式可以避免不同模块的开发者重名冲突的问题。

为了让 Python 将目录当作包，**必须**在该目录下包含__init__.py 文件。在最简单的情况下，__init__.py 可以是一个空文件。

在 Python 3 中，统一使用绝对路径引入包。

2.5.18.9 在 REPL 中重新加载模块

在代码开发测试阶段，笔者经常在 REPL 中加载开发中的模块做测试。在 REPL 中 import（导入）一个模块后，如果该模块在 REPL 外部修改过，则需要退出 REPL 后再进入 REPL，重新加载一次该模块，才能够运行用户程序。我们可以在 REPL 中使用其他方式来重新加载模块。假设我们开发的模块名称为 devmod.py。

```
>>> import devmod
>>> del devmod.py
>>>
>>> import sys
>>> sys.modules.pop('devmod')
>>>
>>> import devmod
>>>
```

与 import 相反的操作是 del，可以将已经导入的模块从当前命名空间删除。但要彻底将导入模块从系统 Python 模块缓存中删除，需要使用 sys 包，找到 sys.modules 即模块缓存对象。使用字典类型的 pop/del 方法删除 sys.modules 的 trigger key。最后重新导入一次 devmod 模块。也就是说，在已经导入 sys 的前提下，删除一个模块，需要两个指令，才能够再次加载该模块。

2.5.19 输入/输出

I/O 在计算机中主要涉及数据在 CPU 与外部设备间的交换，如磁盘、网络等。输入/输出的重点本质上是高速 CPU 与相对慢速的外部接口之间如何进行匹配的问题。输入/输出的解决方案有同步堵塞型和异步事件触发型。围绕输入/输出，硬件设计、操作系统、编程语言、编程模式提供了大量的设计和解决方案。一般初学某种语言都会从同步方法开始学习，除非这种语言天生就是异步的。

2.5.19.1 读写文件

读写文件是最常见的操作之一。读者需要对于文本读写与编码、二进制文件读写、读写模

式和如何及时关闭文件有所了解。

1．读写模式

Python的文件读写方法与C是类似的。open返回文件对象。其最常用的方式是open(filename, mode)。其中，常见模式参见表 2-5 所列。

表 2-5　文件读写模式

字　符	说　明	前　提	位　置	其　他
r	只读文件，只读	必须存在	文件头	
r+	可读写文件，读写	必须存在	文件头	保留文件中未覆盖内容
rU	只读文件，只读	必须存在	文件头	支持通用结束符转换
w	只写文件，只写	新建文件或文件清零	文件头	
w+	可读写文件，读写	新建文件或文件清零	文件头	
a	只写文件，只写	新建文件或追加	文件尾	
a+	可读写文件，读写		新建文件或追加	文件尾

注意

- r/w/a 分别代表只读、只写、追加，但是 w/a 还隐含了可能发生的新建文件操作，而“+”号代表更新，需要与第一个控制字符 r/w/a 组合成 r+/w+/a+理解。
- 读写模式的最后一个字符可以附加"b"，如"a+b"之类的，代表二进制数据。由于 Windows 在文本模式下读写“\n”会被转换成“\r\n”，与二进制模式有区别，因此 b 模式对于 Windows 的读写模式有效，而对 Linux 无效。
- 读写模式的第二个字符可以是“U”，代表支持 Universal Newline，即通用结束符。U 和+不能同时存在。

2．读文件

在前面提到的模式中，一些模式需要文件必须存在；否则会抛出 IOError 错误。

```
>>> open('temp.txt','r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'temp.txt'
```

所以，这种操作需要采用 try...except...finally 来捕获这种异常。

```
>>> try:
...     f = open('temp.txt','r')
... except IOError, e:
...     print(e)
... finally:
...     if f:
```

```
...       f.close()
...
[Errno 2] No such file or directory: 'temp.txt'
```

采用 with 语句可以确保在打开读取文件后，可靠地关闭文件。不过捕获异常依然依靠用户代码。

```
>>> with open('temp.txt','r') as f:
...     print f
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'temp.txt'
```

打开文件后可以使用 read 一次性读入内存，也可以通过 read(length)读取一段指定长度的内容到内存，还可以使用 realines 读取文件并以列表方式进行访问。

```
>>> with open('temp.txt','r') as f:
...     for l in f.readlines():
...         print(l.strip())
...
abcd
1234
Hello,world.
```

3．字符编码

读取文本文件，尤其是非 UTF-8 编码文件时，需要指定其编码的参数。在 Python 3 中，通过 encoding 参数设置文本编码。可以通过 os 来调用 Linux 中的 file 文件以获取文本编码，然后将参数调入。

```
f = open(‘temp.txt’, ‘r’, encoding=’gbk’)
```

在 Python 2.7 中，请使用 io.open。

```
>>> import io
>>> f = io.open('temp.txt','r')
>>> x = f.read()
>>> x
u'abcd\n1234\nHello,world.\n\u4e16\u754c\u4f60\u597d'
>>> print(x)
abcd
1234
Hello,world.
世界你好

>>> f = io.open('temp.txt','rb')
>>> x = f.read()
```

```
>>> x
'abcd\r\n1234\r\nHello,world.\r\n\xca\xc0\xbd\xe7\xc4\xe3\xba\xc3'
>>> print(x)
abcd
1234
Hello,world.
世界你好

>>> f = io.open('temp.txt','r', encoding='utf-8')
>>> x = f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "c:\Python27\lib\codecs.py", line 296, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode byte 0xca in position 26: invalid
continuation byte
```

经过以上的对比交叉实验，如果错误地配置 encoding，解码时就会抛出解码异常。同时注意，encoding 是不可以用于读取二进制文件的。

4．写文件

写文件与读文件在语法上的差别是读写模式字符串的差别。不过操作系统在 write 时会缓存内容，直至 close 方法被调用。所以，使用 with 语句更加保险。

```
>>> with open('temp.txt','a+') as f:
...    f.write('allan\r\n')
...
>>> f = open('temp.txt','r')
>>> f.read()
'abcd\n1234\nHello,world.\n\xca\xc0\xbd\xe7\xc4\xe3\xba\xc3allan\r\n'
```

5．二进制文件

可以通过在读写模式中添加'b'来实现二进制文件的读写。我们来读一下自己模块编译后的 pyc 文件。

```
>>> f = open('class/class_demo.pyc','rb')
>>> f.read()
"\x03\xf3\r\n\xa9E\x87......\x0f\x0b"
```

6．类文件对象

在 Python 中，具备 read 方法的对象都是类文件（file-like）对象，如字节流、网络流、自定义流数据。Python 的这种方式就是鸭子类型。

2.5.19.2　StringIO/BytesIO

上面提到过类文件操作。StringIO 常常被用作字符串缓冲区。StringIO 的特点是其接口和文

件操作是一致的，可以理解为“内存里的文件对象”。我们可以先创建 StringIO 对象，然后像写文件一样写入，并使用 getvalue 方法读取。

```
>>> from io import StringIO
>>> sf = StringIO()
>>> sf.write('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unicode argument expected, got 'str'
>>> sf.write(u'hello')
5L
>>> sf.write(u'world\r\nPython')
13L
>>> sf.getvalue()
u'helloworld\r\nPython'
```

以上说明 Python 2.7 的 io.StringIO 只接受 unicode 字符串。要实现其他编码，需要采用 BytesIO 类。

```
>>> from io import BytesIO
>>> bf = BytesIO()
>>> bf.write('hello world')
11L
>>> bf.getvalue()
'hello world'
```

StringIO 对应的 C 语言版本是 cStringIO。

实际上，StringIO 和 BytesIO 在物联网相关设计中（如 Tornado）是经常被使用的类，但是其却有多个实现。Python 2 和 Python 3 在 StringIO 和 BytesIO 之间有诸多不同。第三方库 six 是一个提供同时兼容 Python 2 和 Python 3 的解决方案。之所以取名为 6，是因为 2×3=6，是 2 和 3 的最小公倍数，取其兼容之意。

这几个模块的具体区别可参考表 2-6 所列的对比表。

表 2-6　StringIO/BytesIO 模块对比

模　　块	Python 2	Python 3
StringIO.StringIO	内存中的字符缓存，unicode/string	N/A
cStringIO.StringIO	C 语言版本，高效但有限制	N/A
io.StringIO	unicode 字符串缓存	文本缓存，不接受 unicode
io.BytesIO	字节缓存	字节缓存
six.StringIO	StringIO.StringIO	io.StringIO
six.BytesIO	io.BytesIO	io.BytesIO

cStringIO.StringIO 是最快的。其在 Python 3.4 中被统一为 io.StringIO。此外，cStringIO 的使用有一些限制：

- cStringIO 不能作为基类被继承。
- cStringIO 不能接收非 ASCII 字符的字符串参数。
- 还有一点与 StringIO 不同的是，当使用字符串参数初始化一个 cStringIO 对象时，该对象是只读的。

io.Bytes 也是通过 C 实现的，但是通过 io.BytesIO(b'data')初始化 BytesIO 对象时会对数据进行一次复制，导致性能上的损失。具体选用哪一种版本，需要考虑对 Python 2/3 代码的兼容性、数据类型（字符串、unicode 或字节串）以及性能要求，选择合适的模块。

2.5.19.3　数据序列化

将程序内的数据结构和对象转换为字符串或字节串等序列类型的过程为序列化，其逆过程为反序列化。

文件读取最简单的是字符串型数据，其他的如数值型或者嵌套列表、字典等带着结构层次的数据，手工解析和序列化会比较耗时。好在有多种方式可以进行这些结构化数据的序列化处理，比如 JSON 和 XML。此外的替代方式还有 msgpack/protobuf 等。其中 JSON 的使用场景逐渐增加并部分替代了 XML，而且 Python 已经将其内置在标准模块中。稍后会专门讲解 JSON/XML 模块。

Python 还有一个模块为 pickle/cpickle。英语中 pickle 指的是腌渍品（如泡菜、腌黄瓜之类的菜品），这用来描述对象序列化还是蛮形象的。作为一种 Python 程序之间的协议，可以将任意复杂的 Python 对象序列化、保存、传递，并反序列化为 Python 对象。但是这种方式有可能带来不安全因素，让不受信任的代码得到执行。此外还有兼容性问题。

2.5.19.4　文件和目录

文件与目录的相关操作集中在 os 和 os.path，以及标准库 shutil 中。

```
>>> import os
>>> os.name
'posix'
>>> os.uname
<built-in function uname>
>>> os.uname()
('Linux', 'Ubuntu-Server', '3.2.0-67-generic-pae', '#101-Ubuntu SMP Tue Jul 15 18:04:
54 UTC 2014', 'i686')
>>> os.path.abspath('.')
'/root'
>>> os.path.join(os.path.abspath('.'),'test')
'/root/test'
```

```
>>> pth = os.path.join(os.path.abspath('.'),'test')
>>> os.mkdir(pth)
>>> os.listdir('.')
['python_lab', 'download', 'test']
>>> os.rmdir('test')
>>> os.listdir('.')
['python_lab', 'download']
```

文件以及目录操作是最常见的操作。需要注意的是 os 和 os.path 的划分方式和我们的使用方式不同。os 是与操作系统有关联的部分，而且一些功能依赖于具体的操作系统而有所不同，比如 uname 方法只在 UNIX 系统中存在。而 os.path 是与路径有关联的部分，凡是和路径操作有关的如 UNIX/Windows 的正反斜杠路径表达，究竟是文件还是路径的判断，以及绝对和相对路径的寻址等都在 os.path 中。但是文件和目录的更改、增加、删除等都归属 os 模块。一些操作如复制、压缩文件等在 os 找不到，需要到 shutil 标准库中寻找。

同时请留意 sys.path 和 os.path 的区别。前者是 Python 的模块系统路径，后者是操作系统的路径操作包。

2.5.20 面向对象编程

面向对象最重要的是类（Class）和实例（Instance）。对象是客观事物的抽象，类是对象的抽象。对象是类的实例，类是对象的模板。两者如同铸件与模具的关系。Python 类提供面向对象编程中的所有标准特点：允许继承多个基类；派生子类可以重载基类的任何方法；对象可以包含任意数量和任意种类数据。Python 作为动态语言，其模块、类在运行时创建，创建后可以进一步修改。

2.5.20.1 类和实例

在 Python 中，定义类采用 class 关键字。而实例是通过“类名（+参数）”创建的。

经典类：

```
>>> class Employee:
...     pass
...
>>> Employee
<class __main__.Employee at 0x00C1EA40>
>>> type(Employee)
<type 'classobj'>
>>> allan = Employee()
>>> allan.__class__
<class __main__.Employee at 0x00C1EA40>
>>> type(allan)
<type 'instance'>
```

新式类：

```
>>> class Employee(object):
...     pass
...
>>> Employee
<class '__main__.Employee'>
>>> type(Employee)
<type 'type'>
>>> allan = Employee()
>>> allan.__class__
<class '__main__.Employee'>
>>> type(allan)
<class '__main__.Employee'>
```

在以上 Python 代码中，有一种类没有定义其继承的父类，而另外一种则显式继承自 object 类。这就是旧式类（又称经典类）和新式类定义的区别。两种类的类型是不一样的。Python 2 默认都是经典类，Python 3 默认都是新式类。请逐渐过渡到新式类。

新式类的好处如下：

- 统一了类（class）和类型（type），类的实例无论是__class__还是 type，返回值都是类的名称。
- 引入更多的内置属性。
- 引入描述符。
- 属性可以计算。

在本节中我们使用以下的新式类定义作为案例。

```
>>> class Employee(object):
...     def __init__(self, name, age=18, salary=3000):
...         self.name = name
...         self.age = age
...         self.salary = salary
...
>>> allan = Employee('allankliu',40)
>>> kirin = Employee('kirin', 6, 1000)
```

与普通函数定义不同，类的定义函数即构建函数是__init__。不过，__init__并不是必需的。空白类仅需 pass 即可。__init__的第一个参数必须是 self，指向实例本身。剩余参数可以使用默认参数来定义。在创建实例的时候，就不能传入空的参数了，除非采用了默认参数。必须传入与__init__方法匹配的参数，但不需要传 self，Python 解释器自己会把实例变量传进去。

Python 的实例还可以动态绑定数据，即同一类的不同实例可以包含不同的变量。

```
>>> allan.role='software'
>>> allan.role
```

```
'software'
>>> kirin.role
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Employee' object has no attribute 'role'
```

在本例中，name 之类的为类属性，而 role 则是实例属性。注意类属性和实例属性不要使用相同名称，因为实例属性会屏蔽掉类属性。

2.5.20.2 访问限制

虽然类“封装”了数据和方法，但 Python 是没有 public/private 的变量区分的。Python 的类属性实际上在外部是可以访问并修改的。这一点在安全性上需要考虑。

```
>>> allan.age = 100
>>> allan.age
100
```

在实际工程中许多属性都不可以修改。在 Python 中，可以使用连续下画线的变量名定义（如__var），将其变成私有变量，约束外部访问。同时，通过定义相关方法来实现这些私有变量的访问。最重要的一点是，在这些方法中可以对于输入变量的范围进行验证。

```
>>> class Employee(object):
...     def __init__(self, name, age=18, salary=2000):
...         self.__name = name
...         self.__age = age
...         self.__salary = salary
...     def getAge(self):
...         return self.__age
...     def setAge(self, age):
...         if 18<age<60:
...             self.__age = age
...         else:
...             print("invalid parameter\r\n")
...
>>> allan = Employee('allan')
>>> allan.__age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Employee' object has no attribute '__age'
>>> allan.setAge(6)
invalid parameter
>>> allan.setAge(35)
>>> allan.getAge()
35
```

此外，Python 的动态语言决定了实际上可以通过某些方法，如 allan._Employee__age，即所

谓的 name mangling 技术来访问甚至修改私有变量。但是不推荐这样做。所以，这仅仅是一个约定而已。

请不要将私有变量的双画线定义__var 与前后双画线的特殊用途变量__var__混用。还有，_var 单画线的用法是模块内的私有变量，在使用 from abc import *时会作为私有变量，无法导入和访问，但是在同一模块内可以访问。此外采用__var 的方式，其子类也无法继承或访问这个变量。

一般 Python 的类中变量名称不推荐带下画线。

2.5.20.3　继承和多态性

在 OOP 设计中，类的不断扩展是通过继承而实现的。在新式类定义中，类都是从 Object 类或者其子类继承而来的。子类从父类（也被称为基类、超类）可以获得父类的私有变量和各类方法。

```
#!/usr/bin/env python

class Employee(object):
    def __init__(self, name, age=18, salary=3000, role='basic'):
        self.name = name
        self.age = age
        self.salary = salary
        self.role = role

    def getAge(self):
        return self.age

    def setAge(self, age):
        if 18 < age < 60:
            self.age = age
        else:
            print("invalid parameter\r\n")

    def transport(self):
        print("Take shuttle bus")

class Executive(Employee):
    def __init__(self, name, age=30, salary=30000,role='GM'):
        Employee.__init__(self, name, age, salary, role)

    def makeBizPlan(self, plan):
        print("{1} {0} says, Out biz plan is {2}".format(self.name,self.role,plan))

    def transport(self):
        print("Company pick-up")
```

```
class SoftwareEngineer(Employee):
    def __init__(self, name, age=25, salary=5000,role='Engineer'):
        Employee.__init__(self, name, age, salary, role)

    def writeSoftware(self, lang):
        print("{1} {0} says, I can write {2}".format(self.name, self.role, lang))

michael = Executive('michael')
michael.makeBizPlan('10 billion sales')
michael.transport()
david = SoftwareEngineer('david')
david.writeSoftware('Python, Java, C++')
david.transport()
```

在以上例子的子类 Executive 和 SoftwareEngineer 的构建函数中，必须显式地调用其父类的__init__方法。例子中父类的变量必须定义为 self.name；如果定义为 self.__name，子类将无法继承该私有变量。

子类 Executive、SoftwareEngineer 继承了 Employee 的各个方法，包括 transport。Executive 还重载了其父类的 transport 方法，其实现与父类和其他子类不同，这就是 OOP 的多态性。多态性还体现在各个子类还有各自独特的方法，如 makeBizPlan 和 writeSoftware。所以，继承+重载=多态性。

2.5.20.4 获取对象信息

Python 中的一切皆为对象。可以采用内置函数 type 来获取对象类型。

```
>>> type(123)
<type 'int'>
>>> type('abc')
<type 'str'>
>>> def fun(a):
...     pass
...
>>> type(fun)
<type 'function'>
>>> type(u'abc')
<type 'unicode'>
>>> type(b'abc')
<type 'str'>
>>> class b(object):
...     pass
...
>>>
>>> type(b)
<type 'type'>
```

对于类的查询可以使用 issubclass 和 isinstance 来判断父类信息。

```
>>> print(isinstance(david,SoftwareEngineer))
>>> True
>>> print(isinstance(david,Employee))
>>> True
>>> print(issubclass(SoftwareEngineer,Employee))
>>> True
```

配合 getattr、setattr 以及 hasattr，还可以直接操作一个对象的状态。

```
>>> print(hasattr(david, 'name'))
>>> True
>>> setattr(david, 'name', 'David Ng')
>>> print(getattr(david, 'name'))
>>> 'David Ng'
```

2.5.20.5　实例、静态和类方法

实例方法就是类的实例都能够访问的方法，这就是我们最常见的方法。

静态方法是一种普通函数，位于类的命名空间中，但不对任意实例类型进行操作。其使用装饰器@staticmethod 进行定义。类对象和实例都可以调用静态方法。

```
class Employee(object):
    def __init__(self, name, age=18, salary=3000, role='basic'):
        self.name = name
        self.age = age
        self.salary = salary
        self.role = role

    def getAge(self):
        return self.age

    def setAge(self, age):
        if 18 < age < 60:
            self.age = age
        else:
            print("invalid parameter\r\n")

    def transport(self):
        print("Take shuttle bus")

    @staticmethod
    def helloworld():
        print('Hello there')

michael = Executive('michael')
michael.helloworld()
```

运行结果如下：

```
> Hello there
```

类方法是将类本身作为对象进行操作的方法。类方法使用@classmethod 装饰器定义，其第一个参数是类，约定写为 cls。类对象和实例都可以调用类方法。

```
class Employee(object):
    company="PNX"
    def __init__(self, name, age=18, salary=3000, role='basic'):
        self.name = name
        self.age = age
        self.salary = salary
        self.role = role
        cost = self.salary
        print cost

    @classmethod
    def companyName(cls, time):
        print('{0}'.format(cls.company))


michael = Executive('michael')
michael.companyName(1)
```

Python 的方法并不像其他语言严格地区分静态方法和实例方法。也就是说 Python 的静态方法、类方法和实例方法只是在调用上有区别，类型和实例都可以调用；区别在于调用参数而已。

2.5.20.6 @property

我们知道类和实例会有自己的属性，但是属性过于灵活，需要额外定义方法实现对于属性的检查。但是采用函数来检查属性略显复杂。可以采用装饰器@property 将方法转换为属性。以下例子通过@property 将类方法转化为只读属性，并且重新实现了属性的 setter 方法。

```
class Employee(object):
    def __init__(self, name, age=18):
        self.name = name
        self._age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, age):
        if 18<age<60:
            self._age = age
```

```
            print("{0} year old".format(self._age))
        else:
            print('Valid range is 18~60')

allan = Employee('allankliu')
allan.age = 30
allan.age = 5
```

运行结果如下：

```
30 year old
Valid range is 18~60
```

2.5.20.7　多重继承

在许多情况下，某些类需要具备多个父类的功能，这时候需要采用多重继承。Java 只允许单一继承，必须通过其他方式来实现多重继承；而 Python 支持多重继承，实现起来方便许多。

不仅仅是第三方库，Python 的标准库里也有大量多重继承的例子。我们可以只用最简单的方法来实现多重继承，如：

```
class whale(Mammal, Swimmable):
    pass
```

可以使用 MixIn 来明确继承主线，如：

```
class dolphin(Mammal, SwimmableMixin,EchoMixIn):
    pass
```

即海豚类继承自哺乳类，同时继承了游泳类和回声类 MixIn 两个类。MixIn 的目的是为一个类增加多个功能。在互联网应用 REST 中最常见的对应于 URL 路径的某个类，继承 JsonrpcRequestHander 中需要的 HTTP 返回方法，同时继承数据库 MixIn 类存取数据方法。

```
class JsonAPIRealtimeHandler(cyclone.jsonrpc.JsonrpcRequestHandler, \
        storage.DatabaseMixin):
    def get(self, device):
        pass
```

2.5.20.8　定制类

在 Python 中的__xyz__，即前后双下画线方式定义的都是特殊变量或函数名。类定义中的__init__就是最常见的例子。本节介绍另外一个常见的方法：__str__。

通常打印一个自定义类的对象，不会很直观。

```
>>> class Employee(object):
...     def __init__(self, name, age=18):
...         self.name = name
...         self.age = age
```

```
...
>>> allan = Employee('allankliu')
>>> print(allan)
<__main__.Employee object at 0x00D9E050>
```

如果需要针对自定义类实例进行有意义的打印，可以使用__str__方法。

```
>>> class Employee(object):
...     def __init__(self, name, age=18):
...         self.name = name
...         self.age = age
...     def __str__(self):
...         return 'Employee object: {}, age of {}'.format(self.name, self.age)
...
>>> allan = Employee('allankliu')
>>> allan
<__main__.Employee object at 0x00D93FD0>
>>> print(allan)
Employee object: allankliu, age of 18
```

__str__影响到的是 print 函数，而__repr__则会影响到 repr 函数（可以另外定义）。

本节仅是简单介绍，若涉及类的定制，则请读者参阅官方文档以了解更多详情。

2.5.20.9 元类编程

正常情况下，我们都用 class 来定义类。type 函数也允许动态创建类。动态语言本身支持运行时动态创建类，这和静态语言有非常大的不同。

metaclass，元类，简单的解释就是：使用元类创建类，然后使用类创建实例。通常来说，开发者不会接触到元类编程。但是当可以定制对象创建的过程时，许多新的编程构造变得更容易，或者成为可能。元类支持某些类型的“面向切面编程”，例如，用一些特性来增强类、跟踪能力、对象持久性、异常日志记录以及其他特性。其主要用于数据库、日志等相关场合。

在 9.7.10 节提到的 Cyclone Web 框架中，也是用元类编程来简化数据库接口的。

元类是很复杂的。对于非常简单的类，不希望通过使用元类来对类做修改。可以通过其他两种技术修改类：

- Monkey patching；
- 类装饰器。

当需要动态修改类时，可以使用上面这两种技术解决大多数问题。例如，在许多 Web 框架中，大多使用装饰器来解决面向切面的权限访问需求。

2.5.21 进程和线程

现在的操作系统都是多任务操作系统。即使是嵌入式操作系统里也都采用 RTOS（实时操

作系统）进行任务调度。但这些“多任务”都是通过将时间分片轮流分配给各个任务而实现的。真正的并行任务是基于多核 CPU 架构来实现的。无论是多核还是单核，操作系统都会把任务轮流调度到每个核心上执行。在 9.5.1 节中会提到并发（concurrency）和并行（parallelism）的差异。

对于操作系统来说，一个任务就是一个进程（process）。有的进程比较复杂，需要同时运行多个内部任务，这些子任务可以被称为线程（thread）。进程和线程的联系和区别如下。

- 地址空间：即进程内的执行单元，多个线程可共享进程的地址空间。
- 资源拥有：进程是资源分配和拥有的单位，同一进程内的线程共享资源。
- 最小调度：线程是处理器和操作系统的调度基本单位，进程则不是。
- 两者均可以并发执行。

操作系统不会将多个线程看成单独运行的应用来实现调度管理和资源分配。线程更多的是应用程序根据应用特点划分出的可并行处理的部分。

多进程和多线程的解决方案如下：

- 启动多个进程，多个进程可以一起执行任务。
- 启动单一进程，在进程中启动多个线程，由多个线程执行多个任务。
- 多进程再启动多线程，管理比较复杂。

Python 有支持多进程和多线程的标准模块。但无论哪一种，都涉及同步、数据共享问题，设计上需要仔细规划。

2.5.21.1　GIL

Python，尤其是 CPython 有个臭名昭著的限制：GIL（Global Interpreter Lock，全局解释器锁定）。这号称是 Python 的最大难题。

Python 解释器并非是线程安全的设计。为了支持多线程 Python 程序，设置了 GIL。当前线程访问 Python 对象之前必须锁定。如果没有这个锁，在多线程程序中，即便最简单的操作也会造成问题，尤其是两个线程同时操作同一对象的时候。

所以，仅有获得 GIL 的线程才可以操作 Python 对象或者调用 Python/C API 函数。为了模拟并发执行，解释器会执行一段字节码后，定期切换线程。该锁定也会在堵塞式 I/O 操作（如文件读写）时释放掉，让其他 Python 线程可以同时运行。

由于 GIL 的存在，CPython 中并不存在真正意义上的多线程。这一点是 CPython 弱于 C/C++/Java 等语言的一点。通常使用 Cython/Jython 等运行时来实现多线程。

2.5.21.2　多进程

多进程设计可以允许 Python 程序充分利用多核处理器甚至多台分布式计算机的处理能力。早期的 Apache 服务器是采用多进程设计的，每接到一个新的 HTTP 请求，就 fork 一个新的进程。但是多进程相对耗费资源。现在的许多架构往往是混合模型。

前面提到多进程与操作系统有很大关联，不同的操作系统间存在着差异。反映在 Python 语言的多进程模块上，UNIX/Linux 采用 os.fork。此外，还提供了 multiprocessing 跨平台模块以增加对于 Windows 的多进程支持。

UNIX/Linux 提供 fork 系统调用，调用一次 fork，操作系统就把当前进程（父进程）复制一份（子进程），然后分别在父进程和子进程返回。子进程返回 0，父进程返回子进程 pid。

在 UNIX/Linux 中可以使用 os 模块来访问这些服务。

```
import os
print('Process (%s) start...'%os.getpid())
pid = os.fork() # returned pid
if pid == 0:
    print('child process (%s) from parent process (%s)'%(os.getpid(), os.getppid()))
else:
    print('current process (%s) created child process (%s)'%(os.getpid(), pid))
```

跨平台 multiprocessing 模块的使用方法如下：

```
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

如果需要批量产生进程，可以使用 multiprocess.Pool。

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)
    result = pool.apply_async(f, [10])
    print result.get(timeout=10)
    print pool.map(f, range(10))
```

Subprocess 可以用于产生子进程。

```
>>> subprocess.call(['ls','-l'])
total 1
drwxr-xr-x    2 Allan    Administ       0 Jul 14 22:02 more_demo
drwxr-xr-x    3 Allan    Administ       0 Jul 17 09:26 multiprocessing_demo
-rw-r--r--    1 Allan    Administ      42 Jul 16 17:43 temp.txt
```

```
drwxr-xr-x    2 Allan    Administ        0 Jul 16 08:42 local_demo
0
>>>
```

进程之间可以采用 Queue 和 Pipe 进行通信。Queue 队列是一种通用的 FIFO 结构，命名空间内的对象都可以访问。而 Pipe 管道是一种双向通信的机制，采用收发两端进行。

queue_demo.py：

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print q.get()    # prints "[42, None, 'hello']"
    p.join()
```

pipe_demo.py：

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print parent_conn.recv()   # prints "[42, None, 'hello']"
    p.join()
```

2.5.21.3　分布式进程

进程不仅仅可以实现并发（concurrency），还可以实现并行（parallel）。进程可分布在不同 CPU 核心或者网格计算的多台计算机中，实现空间上的同时执行。并行的执行往往依赖于硬件在不同层面的冗余，如采用多级流水线的 CPU，采用多个 CPU 核心的对称多处理（SMP），采用众多处理单元的 GPU，采用多台服务器的云计算网格，采用众多逻辑单元的 FPGA，等等。本节只介绍在多 CPU 核心与网格计算中的分布式进程。

与线程相比，进程更加稳定，而且可以分布在不同机器上运行。而线程只能够在同一台机器的多个 CPU 核中运行。Python 中的 multiprocessing.managers 模块支持将多进程部署到不同机器上。一个服务进程作为调度，将任务分配到不同机器中，彼此依靠 TCP/IP 网络通信。这种方

式很容易编写分布式进程应用。

假设现有通过 Queue 通信的多进程程序在同一台机器上运行。现在增加一台机器，并分别部署发送任务和处理任务。可以采用分布式进程。我们依然可以采用 Queue 进行异步通信，并通过 managers 将 Queue 网络端口暴露给需要部署的其他机器。

做实验时，可以在同一台机器上模拟两个进程，也可以在虚拟机中运行另外一个进程与主机进行通信。

server_process.py：

```
from multiprocessing.managers import BaseManager
import Queue
queue = Queue.Queue()

class QueueManager(BaseManager):
    pass

QueueManager.register('get_queue', callable=lambda:queue)
m = QueueManager(address=('', 50000), authkey='mykey')
s = m.get_server()
s.serve_forever()
```

client_put_process.py：

```
from multiprocessing.managers import BaseManager

class QueueManager(BaseManager):
    pass

QueueManager.register('get_queue')
m = QueueManager(address=('localhost', 50000), authkey='mykey')
m.connect()
queue = m.get_queue()
print "put info into queue"
queue.put('hello')
```

client_get_process.py：

```
from multiprocessing.managers import BaseManager

class QueueManager(BaseManager):
    pass

QueueManager.register('get_queue')
m = QueueManager(address=('localhost', 50000), authkey='mykey')
m.connect()
queue = m.get_queue()
print(queue.get('hello'))
```

在实际工程设计中，往往会采用专门的队列服务：消息队列，并构成微服务结构，通过容器技术分发。

2.5.21.4　多线程

高级语言通常支持多线程。Python 线程在 UNIX 中是 POSIX 线程，而非模拟出来的多线程。Python 标准库中有底层多线程模块_thread 和高层多线程模块 threading。通常，开发中使用 threading 模块。

将函数名传入并创建 Thread 实例，调用 start 方法。

```
import time, threading

def loop():
    print('thread %s is running...' % threading.current_thread().name)
    n = 0
    while n < 5:
        n = n + 1
        print('thread %s >>> %s' % (threading.current_thread().name, n))
        time.sleep(1)
    print('thread %s ended.' % threading.current_thread().name)

print('thread %s is running...' % threading.current_thread().name)
t = threading.Thread(target=loop, name='LoopThread')
t.start()
t.join()
print('thread %s ended.' % threading.current_thread().name)
```

运行结果如下：

```
thread MainThread is running...
thread LoopThread is running...
thread LoopThread >>> 1
thread LoopThread >>> 2
thread LoopThread >>> 3
thread LoopThread >>> 4
thread LoopThread >>> 5
thread LoopThread ended.
thread MainThread ended.
```

每个进程都会启动一个线程，即主线程。该线程可以启动子线程。threading 模块的 current_thread 函数返回当前线程实例，主线程 name 属性为 MainThread，子线程的名字则为 Thread-1/2……不过也可以单独自定义命名，本例中为 LoopThread。

2.5.21.5　线程资源锁定（Lock）

多线程与多进程的最大区别在于：多进程中的每个进程会拥有自己的变量，互不影响；而

多线程中的所有线程共享变量。任何一个变量都有可能被多个线程修改。这种资源竞争会造成数据的不一致性。所以，需要一种互斥布尔量来锁定需要保护的变量。

当某个线程要访问某个共享对象时，需要使用 threading.Lock().aquire()方法获取 Lock，访问后需要释放锁定：threading.Lock().release()。当多个线程竞争这个变量时，有且仅有单一线程可以锁定变量并继续执行，其他线程必须等待，直至获得变量锁。

```
#!/usr/bin/env python

import threading
import time
import os
import random

total_ticket = 50
lock = threading.Lock()

def worker():
    #print "time wasting job\r\n"
    time.sleep(random.random()*2)

def ticketBooth(tid):
    global total_ticket, lock
    while True:
        lock.acquire()
        if total_ticket:
            total_ticket -= 1
            print "total ticket: %d, by %d, %s\r\n"%(total_ticket, tid, threading.
current_thread().name)
        else:
            print "ticket sold out\r\n"
            os._exit(0)
        lock.release()
        worker()

print "total ticket: %d, start to sell\r\n"%(total_ticket)
for i in range(10):
    t = threading.Thread(target=ticketBooth, args=(i,))
    t.start()
```

在线程模型中，还有其他的互斥变量，请读者自行参考文档。

在嵌入式系统中，虽然不一定有多线程，但有可能出现中断服务和主循环同时修改（临界区：Critical Section）变量的现象。因此，也需要类似的保护机制。

2.5.21.6　Thread local

线程修改全局变量必须加锁，比较麻烦。线程局部变量作用域局限在线程本身，不会影响到其他线程。但在函数调用的时候，传递局部变量很麻烦。采用 Thread local 可以很好地解决这个参数传递的问题。Thread local 来自 threading.local 模块。

Thread local 在 Web/IoT 中的常见应用如下：将 HTTP/TCP/UDP 请求、数据库连接、设备串口、API 认证信息、用户身份信息作为 Thread local，这样线程的数据处理流程中的函数可以很容易地访问这些局部变量属性值。虽然 Thread local 是全局变量，但是其保存的却是与每个线程有关的局部变量，线程可以访问各自的局部变量而互不影响。

```
import threading
import random

local = threading.local()
local.tname = 'main'
alias = ['allan','kirin','python','michael','jackson','janet']

def lowerfunc():
    local.alias =  random.choice(alias)

def func(x):
    local.tname = 'thread-{}'.format(x)
    lowerfunc()
    print('{}:{}'.format(local.tname, local.alias))

for i in range(10):
    th = threading.Thread(target=func, args=str(i))
    th.start()
    th.join()

print local.tname
```

运行结果如下：

```
thread-0:allan
thread-1:michael
thread-2:kirin
thread-3:jackson
thread-4:kirin
thread-5:allan
thread-6:python
thread-7:allan
thread-8:michael
thread-9:allan
main
```

2.5.21.7 协程

协程，英文为 coroutine，是 co-operative routine 的缩写。不同语言对于协程的支持程序是不一样的。Python 中可以使用 yield 语句的生成器，或者其他第三方库 gevent 来实现。

2.5.22 错误和异常

Python 中有两种常见错误：语法错误和异常。

2.5.22.1 语法错误

语法错误往往是用户代码中存在的语法的解析错误。比较典型的是少了括号和冒号、多个分号之类的问题。

2.5.22.2 异常

异常是语法正确的代码在运行时检测到的错误，比如除数为零、未导入模块、类型错误，等等。Python 有一系列的内置异常，这些异常可以在相关文档中找到。也可以实现用户自定义异常。

2.5.22.3 处理异常

用户代码可以捕获并处理异常。Python 中使用 try...except...finally 来实现其目的。较为完整的例子如下：

```
>>> def divide(x,y):
...     try:
...         res = x/y
...     except ZeroDivisionError:
...         print "Div by Zero!"
...     else:
...         print "res is",res
...     finally:
...         print "finally clause"
...
>>> divide(2,1)
res is 2
finally clause
>>> divide(2,0)
Div by Zero!
finally clause
>>> divide("2","10")
finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
```

```
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

2.5.22.4　引发异常

raise 语句允许程序强制引发异常。raise 的唯一参数是异常，可以是异常实例或者异常类（Exception 子类）。

```
>>> raise NameError('You hit it!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: You hit it!
```

用户主动强制引发异常主要用于终止迭代，或调试需要。

2.5.22.5　用户自定义异常

用户可以创建异常类来自定义异常，异常继承自 Exception 类。

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return repr(self.value)

try:
    raise MyError(100)
except MyError as e:
    print "My exception occured, value", e.value

raise MyError('Oh!')
```

当我们在设计一个较为复杂的模块时，需要先定义可能引发的不同异常。为模块定义的异常创建基类，并为不同错误情况创建特定子类。

```
class Error(Exception):
    pass

class InputError(Error):
    """
        expr -- input expression in which the error occurred
        msg  -- explanation of the error
    """
    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg

class TransitionError(Error):
    """
```

```
    Attributes:
        prev -- state at beginning of transition
        next -- attempted new state
        msg  -- explanation of why the specific transition is not allowed
    """
    def __init__(self, prev, next, msg):
        self.prev = prev
        self.next = next
        self.msg = msg
```

2.5.22.6 清理操作

try 语句中可选的 finally 子句，用于在所有情况下必须执行的操作，即清理操作。

```
>>> try:
...     raise NameError('An error!')
... finally:
...     print("last clean-up")
...
last clean-up
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: An error!
```

无论是否发生异常，finally 子句都会执行。本例中没有 except，会在 finally 子句之后重新引发异常。在 if...elif...else 与 try...except...finally 混用的时候，需要注意其各自执行的流程。

在网络连接中，finally 可以用于释放外部资源（如文件或网络连接），无论之前是否成功。

2.5.22.7 预定义清理操作

在前面提到的上下文管理器类型中，提到过 with 语句。在文件读取等场景中，最后必须关闭文件，但这一过程可能因为各种异常而中断。with 可以帮助简化这些情况。

文件打开是常见的操作：

```
for line in open('myfile.txt'):
    print line,
```

这个例子执行后，还会让文件在未知的一段时间内保持打开状态。这在大型设计中是很危险的。而采用 with 语句可以确保文件对象最后被立即清理掉。

```
with open('myfile.txt') as f:
    for line in f:
        print line,
```

采用 with 语句后，myfile.txt 总会被关掉，即使执行体内出现错误或异常也会将其关闭。关于 with 语句的工作原理、背景、例子，请查看 Python PEP0343 文档。

2.6　Python 标准库概览

Python 的标准库种类繁多，涵盖多个方面。这也是 Python 生态的最重要组成部分，值得初学者花时间好好掌握。

Python 标准库包含如下内容：

- 字符串服务；
- 文本服务；
- 二进制服务；
- 数据类型；
- 数值与数学模块；
- 函数式编程；
- 文件和目录访问；
- 数据持久化；
- 数据压缩和归档；
- 文件格式；
- 加密服务；
- 操作系统通用服务；
- 操作系统可选服务；
- 进程间通信和网络通信；
- 互联网数据处理；
- 结构化标记语言处理；
- XML 处理模块；
- 互联网协议支持；
- 多媒体服务；
- 国际化和本地化；
- 程序框架；
- 图形界面 Tk；
- IDLE 界面；
- 开发工具；
- 调试和剖析；
- 性能分析工具；
- 软件包和分发工具；
- Python 运行时；

- 定制 Python 解释器；
- 模块导入；
- Python 语言服务；
- Windows 相关服务；
- UNIX 相关服务。

学习使用标准库最简单的方式是查阅文档，还可以查阅标准库源代码了解其实现。Python 标准库的源文件路径如下。

- Windows：C:\Python27\libs*.py（不包含第三方模块 site-packages）。
- Linux：/usr/lib/Python2.7/*.py（第三方模块保存在/usr/local/lib/Python2.7/site-packages 目录中）。

2.7 本章小结

本章主要介绍了 Python 的基础知识和标准库。尤其是 Python 内置类型、内置函数以及相关数据结构，值得读者仔细揣摩，并充分利用于物联网的开发中。至于标准库，则承担着辞典的作用，读者在应用开发时要记得经常查询，以免闭门造车。

第 3 章

Python 语言进阶

了解了 Python 语言的基础知识后，我们需要了解 Python 在物联网应用开发中经常使用的技巧和必须掌握的进阶知识。尤其是许多开发框架已经大范围使用了 Python 的语法糖。这些设计对于了解其内情的开发者来说很实用，但是对于刚入门的读者来说可能需要了解其背后实现的工作原理。所以，本章会展开介绍这些内容。

3.1 HOWTO：常见任务和解决方案

物联网以解决设备联网为第一任务，但实质上均围绕数据进行处理。从最前端的传感器进行数据采集，然后逐层传输、处理，并在服务器端进行数据分析、可视化，形成对组织和业务有意义的应用。以数据传输链条为主线，物联网中的常见任务如下。

- 数据采集：从 ADC、GPIO 和各类总线读取传感器数据。数据采集依赖于硬件，需要从硬件寄存器中读取数值，其宽度为字节、字、双字，即 8~32 位宽无符号整数，也可以经过初步转化后变成浮点数和序列化对象。
- 数据缓存：任何数据从一个实体传输到下一个实体，往往需要将数据进行缓存。这些均在系统 RAM 中，包括内存型数据库（如 Redis）中实施。
- 数据多路复用和解复用：主要用于将多组数据打包后在通信通道中传输。
- 数据序列化和反序列化：在传输前将数据对象串行化，传输后解析为数据对象。
- 数据压缩和解压缩：为了减少冗余数据对于通道带宽的占用而实现的压缩和解压缩。
- 数据加密：通过各种加密算法实现安全通道。
- 数据传输：通过各种协议将数据传输给对等实体。
- 数据后处理：主要指对于原始数据进行类型转换、缩放、归一化，以及频域分析等。
- 数据持久化：RAM 是有限的，适合热点数据，必须将数据持久化到磁盘中，用于之后的数据分析。

- 数据检索：通过某种手段如数据库系统，对已有数据进行检索。
- 数据分析：统计、融合、挖掘，对原始数据进行统计分析，合并其他数据，提炼出内在模式，挖掘出商业价值。
- 数据可视化：将原始数据、统计数据以及数据模式，通过图表、动画形式提供给用户。
- 数据转发：包括推送、异步通知、分享等都归类于此。

前面罗列了 Python 的内置类型、内置函数、类、实例、模块和标准库。本节则立足这些 Python 基础构件，针对从数据采集到数据持久化的物联网需求提供最常见的解决方案。但是从数据检索到数据转发，往往涉及后端设计以及某些专业领域的分析处理，需要第三方扩展模块、包和系统框架，这些内容会在第 9 章、第 10 章中详细介绍。

3.1.1 数据类型转换

数据采集后的寄存器二进制整数如何成为有意义的数据呢？必须做类型转换。鉴于 Python 和 C/C++联系密切，所以我们有必要将两者进行比较。在 C/C++中，程序从寄存器读取的数值都是各种宽度的无符号整数：

- unsigned char，8 位；
- unsigned int，16 位；
- unsigned short，依赖于平台；
- unsigned long，依赖于平台。

所谓依赖于平台指的是与 CPU 架构和编译器有关。成熟的 C 程序，往往会有一个 typedef.h 来确定 unsigned short 和 unsigned long 的宽度到底是多少位。typedef.h 中使用含义明确的 uint8_t、uint16_t、uint32_t 和 uint64_t 来明确宽度和整数范围以避免歧义。通信通道中倾向于以 unsigned char 类型传输，并需要定义字节次序，即所谓大端和小端。也就是将更宽位数数值切割成 8 位传输。由于 C/C++中 unsigned char 也是 8 位整数，可以进行数值计算，所以该类型既可以传输也可以计算。

不仅仅是 C 语言，Python 在通信通道上传输的数据类型也是字符串类型。然而，Python 的字符串类型可以传输，但不能够直接进行数值计算，这和 C/C++完全不同。此外，Python 没有 8 位字节宽度整数，这一点与 C/C++区别很大。

```
>>> a = 'x'
>>> a/2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>> ord(a)/2
60
```

如果将字符串除以 2，Python 会抛出类型错误：除法操作符不能够用于字符串和整数类型之间。必须使用 ord 内置函数取出字符'a'的索引数值，才可以进行计算。所谓索引数值即字符'a'在 ASCII 码表中的下标：0x78，也就是整数 120。该整数除以 2 的结果是 60。所以 Python 虽然是动态类型语言，但涉及通信相关序列化，需要进行显式的转化。

类型转换的方法和模块如下：

- 各内置类型的构建函数。
- struct 模块在序列化和反序列化的同时进行类型转换。
- binascii 模块中的某些方法。

数据序列化、数据显示打印往往依赖于数据类型转换，这些内容会在后面逐一介绍。

3.1.1.1　string/unicode/bytes

在物联网应用中，传输和处理均基于二进制数值和字符串。但是由于计算机发展历史的原因，字符串和二进制数值曾经长期混用。所以，理解 Python 中 string（字符串）、unicode（unicode 字符串）、int（整数）和 bytes（字节串）的概念是非常重要的；而且还要记住，这是 Python 2 和 Python 3 的核心区别之一。具体相关差别参见表 3-1 所列。

表 3-1　Python 2.7/3.X 类型对比

类　　型	Python 2	Python 3
string	默认为 ASCII（8 位）	默认为 unicode（UTF-8）
unicode	默认为 unicode（UTF-8/UCS2）	N/A
int	int 和 long 有符号整数	默认为 long 有符号整数
bytes	等同于 string	单独 bytes 类型

在 Python 2 中，所有采集数据的 int 类型需要转化为 string 类型做传输，传输后转化为 int 做数学计算。unicode 在传入英文时可以和 string 互换，而非英文字符转换需要指定编码或者与平台有关。在 Python 2.7.11 中，bytes 是 string 的别名。也就是说，在物联网应用中，Python 2 以 string 为核心与其他类型进行转换。

在 Python 3 中，string 默认就是 unicode，而二进制数据则变成了 bytes 类型，int 主要和 bytes 进行转换。在物联网应用中，Python 3 以 bytes 类型与其他类型进行转换。

在 Python 3 中，增加了针对二进制数据操作的内置类型：bytes 和 bytearray。可以使用支持缓冲区协议的 memoryview 对象来访问二进制数据而无须复制副本。

3.1.1.2　字节串（bytes）

字节串类型是不可变类型，其由多个字节组成。即每个元素以 8 位保存，取值范围为 0~255。由于在字节编码中 ASCII 占据主导，所以该类型中仅提供了针对 ASCII 数据有效的方法。其声

明方式是：以 b 为前缀，带单个或三个单引号或双引号的类型定义，包括 b'python'、b"python"、和 b'''python'''，其和 string 类型定义的区别就是以 b 来定义。在 Python 2.7 中也有一样的声明方式，但是类型却是字符串。由此可以看出它们演化的过程。所以，在和通信密切关联的设计中，可以逐渐放弃字符串，而使用字节串来定义物理量实现 Python 2/3 兼容。

3.1.1.3 字节数组（bytearray）

字节数组可以被理解为字节串类型的可变版本，即可以就地修改数组中某个字节的内容。在物联网应用中，可以使用预定义的字节数组来定义收发缓冲区。这比用列表实现一维数组要节省资源。在第 6 章中介绍的嵌入式版本 MicroPython/PyMite/Zerynth 中就大量使用了 bytearray 来做缓冲区。

3.1.1.4 数值字符串与数值

所谓数值字符串就是人可以看得懂的数值。如“100”，其实它是字符串，由“1”、“0”、“0”三个字符组成，但是该字符串的含义是整数 100。数值字符串的主要目的是与用户交互，而做真正的计算时需要转换成真正的整数 100，即 0x64。我们先看看整数的转换。

```
int('1010',2) => 10 # 二进制字符串
int('12',8) => 10 # 八进制字符串
int('10') => 10 # 十进制字符串
int('10', 16) => 16 # 十六进制字符串
int('0xFF', 16) => 255 # 十六进制字符串
```

长整型与此类似，其区别在于输出长整型类型整数。

```
long('0xFF',16) => 255L # 十六进制字符串
```

1．整数进制间的转换

```
int(0b1010) => 10
int(012) => 10
int(10) => 10
int(0xFF) => 255
```

2．数值与数值字符串的转换

```
bin(10) => '0b1010'
oct(10) => '012'
hex(10) => '0xa'
```

这里 bin/oct/hex 带入的参数可以是任何形式的整数。

利用 str 也可以产生许多数值类型所对应的字符串。

```
str(0b1010) => '10'
```

```
str(012) => '10'
str(10) => '10'
str(0xFF) => '255'
str(3.1415926) => '3.1415926'
```

3．整数和字符串类型转换

如果仅仅解决整数类型和字符串类型的变换，请使用 ord 和 chr/unichr。请观察这两个内置函数和 int、str 的区别。

```
chr(22) => '\x16'
str(22) => '22'
```

str(x)将输入参数转换为对应的字符串，而 chr(x)将输入参数转换为整数值对应的字符类型。

```
ord('\x32') => 50
int('\x32') => 2
```

int(x)将 0x32 字符转换为 ASCII 中字符'2'的整数 2，而 ord(x)将 0x32 字符转化为整数 50。

我们再观察 chr 和 unichr 的区别。

```
ord('x') => 120
chr(120) => 'x'
ord(u'x') => 120
unichr(120) => u'x'

s = unicode('汉',encoding='GB2312') #在 Windows 7 Python 2.7 REPL 中输入汉字
s = u'u6c49')
ord(s) => 27721
int('6c49',16) => 27721
```

这里，unicode 类型（UCS2 编码）也可以和整数互换类型。

4．浮点数构建函数

与整数不同，float 可以将任何数制的整数或者字符串转化为浮点数。

```
float(2) => 2.0
float('2') => 2.0
float(0xFF) => 255.0
```

5．浮点数与 HEX 字符串转换

```
>>> 3.1415926.hex()
'0x1.921fb4d12d84ap+1'
>>> float.fromhex('0x3.a7p10')3740.0
```

3.1.1.5　类型转换相关内置函数和标准库

上述许多类型转换方法实际上都是 Python 的内置函数。表 3-2 罗列了相关的内置函数。

表 3-2　与类型转换相关的内置函数

函数/方法	说　明
int(x[,base])	将 x 转换为 base 进制的整数
long(x[,base])	将 x 转换为 base 进制的长整数
float(x)	将 x 转换为浮点数
complex(r[,i])	创建 r+ij 复数
str(x)	将对象转换为字符串
repr(x)	将对象转换为表达式字符串
eval(x)	计算字符串中的 Python 运算式，并返回
tuple(x)	将序列转换为元组
list(x)	将序列转换为列表
chr(x)	返回整数参数为地址下标的字符串
unichr(x)	将整数转换为 unicode 字符，chr(x)的 unicode 版本
ord(x)	返回字符串的序号（ordinal），即将字符串转换为地址下标
hex(x)	将整数转换为十六进制 HEX 字符串
oct(x)	将整数转换为八进制 OCT 字符串
bin(x)	将整数转换为二进制 BIN 字符串

以上都是 Python 内置类型的构造函数，如 int、float、str、tuple、chr、ord、hex、oct、bin 等。除了类型转换，更多的是许多不同种类字符串表达和相对应内置类型的转换。例如：浮点字符串和浮点类型转换等。由于 Python 内置类型的多样性，所以这种转换会随着更多类型的引入而不断变化，请读者自行总结。

由于 unicode 的发展历史原因，string/unicode/bytes 等各类方法中都有 encode/decode 的编码指定方法或者参数设置。在 Python 2/3 兼容性设计中，这一点需要读者参考文档，仔细设计与测试。

此外，在稍后介绍的信道序列化和反序列化章节（3.1.7 节）中，我们会大量依赖于 struct 包。struct 的 pack/unpack，除了序列化和反序列化功能，还能够将字符串直接转换为对应的数据类型。其一石二鸟，功能强大，是物联网常用的包。

3.1.2　数据的调试打印

在物联网应用开发中，尤其是适配底层接入协议，抓包解析的时候，需要检查二进制数据通信，此外在调试和日志记录场景中也需要查看二进制数据。而打印整数或者字符串都不合适：整数不直观，字符串容易出现乱码，最好以 HEX 字符串形式打印或者保存。这时候，binascii

标准库很有用。其实，binascii 是一种比较底层的包，包括了针对 CRC32、QP-7bit、base64、UUEncode 等格式的多个转换模块。

3.1.2.1　binascii

binascii 包中的方法 hexlify，其逆方法为 unhexlify，示例如下：

```
>>> import binascii
>>> x = '01234567890'
>>> binascii.hexlify(x)
>>> '3031323334353637383930'
>>> x =  '6162636465666768696A'
>>> binascii.unhexlify(x)
>>> 'abcdefghij'
```

hexlify 其实对应 binascii.b2a_hex，而 unhexlify 对应 binascii.a2b_hex 函数，只不过名字更加直观。

binascii 已经可以满足大部分的二进制调试需求，但是其输出的 HEX 字符串是连在一起的。如果对于显示格式有更高要求，则最简单的方式是使用格式化打印方式：

```
>>> s = 'hello World'
>>> for h in s:
...    print('%02X'%ord(h)),
...
68 65 6C 6C 6F 20 57 6F 72 6C 64
```

读者可以尝试用迭代器+匿名函数或者列表推导式改写该小段代码。也可以将 binascii.hexlify 的 HEX 字符串切片到列表后，再插入空格，采用 join 重新合并成新的字符串。

```
>>> s = 'hello World'
>>> hx = binascii.hexlify(s).upper()
>>> [hx[i:i+2] for i in xrange(0,len(hx),2)]
['68', '65', '6C', '6C', '6F', '20', '57', '6F', '72', '6C', '64']
```

切割成列表后，可以自由发挥各种格式。不过，笔者不太推荐为了调试而使用列表类型，这比较耗费资源。

3.1.2.2　repr 函数

对于一些无法用肉眼查看的对象，推荐使用 repr 函数将其转换为对应字符串表达。

```
>>> a = b'物联网开发'
>>> a
'\xce\xef\xc1\xaa\xcd\xf8\xbf\xaa\xb7\xa2'
>>> repr(a)
"'\\xce\\xef\\xc1\\xaa\\xcd\\xf8\\xbf\\xaa\\xb7\\xa2'"
```

这在一些需要了解字符串编码情况的时候挺有用。

3.1.3 数据类型资源优化

在 Python 的基础知识中，我们知道内置类型分为可变（mutable）类型和不可变类型。两者的核心差别在于，可修改的方法的有无和对于资源的消耗不同。**尽量使用不可变类型**，这样可以避免并发线程和进程同时修改对象的危险。物联网中最常见的模式包括：

- 字符串（string/unicode，不可变）；
- 字节串（bytes，不可变）；
- 元组（tuple，包含统一的字符元素，如（'a','b','b'），不可变）；
- 列表（list，包含统一的字符元素，如['a','b','b']，可变）；
- 字节数组（bytearray，可变）。

总的原则：能够用 tuple/bytes/string/unicode，则不使用 list 和 bytearray。具体使用哪种类型，需要根据 Python 运行上下文环境和版本来定。必要时，我们可以在不同的对等类型间进行转化。

可变类型与不可变类型间的变换

```
tuple([1,2,3]) => (1,2,3)
list((1,2,3)) => [1,2,3]
```

3.1.4 数据结构与算法

数据结构是计算机存储、组织数据的方式，指相互之间存在一种或多种特定关系的数据元素的集合。数据结构研究的是数据元素之间的逻辑关系、物理结构和运算方法。逻辑关系包括集合、线性、树形和图形关系。常见的数据结构包括数组、堆栈、队列、链表、树、图、堆和散列表。

Python 包含了若干种标准的编程数据结构：list、tuple、dictionary、set。其中 list 非常灵活，可以支持数值、堆栈、队列结构。其他的数据结构实现方法如下：

- 用类实现链表；
- 用 defaultdic 实现树；
- 用 heapq 实现堆；
- 用 dict 实现散列表。

接下来，我们看看数据类型的某些说明。

3.1.4.1 数组

当应用中使用大量数据时，使用 array 数组比 list 列表更加有效。因为数组被限定为单一数

据类型，所以其内存使用更加高效。另一好处是，数组可以使用列表一样的方法。所以，使用数组可能对于计算和存储能力受限的系统更加有利。

3.1.4.2　排序

如果应用中需要对有序列表进行数量增减，可以考虑一下 heapq。采用 heapq 函数来增减对象，可以在较小的消耗下维持列表的排序。

另外一种构建有序列表或数值的方式是 bisect。它使用二进制检索寻找插入点，这对频繁修改并重新排序的有序列表是一种很有效的替代方法。

3.1.4.3　队列

队列是常用的数据结构。尽管内置的列表类型可以使用 insert/pop 方法来模拟队列，但这并不是线程安全的。在线程间有序通信应用中，应该使用 Queue.multiprocessing 方法。它包含了一个可以在进程间通信的队列类型。

3.1.4.4　集合容器

集合容器（Collections）包含了若干个数据类型的实现。例如：

- namedtuple，可命名元组，扩展了普通元组，每个成员除了数字索引还增加了命名属性。实际上它是一个函数，而非类型。为元组条目命名，即增加名字属性，可以用于地点命名、地理位置等场景。
- deque 是一种双头队列，可以从两端增减数据。该类型主要用于解决 list 条目增减时性能不足的问题，其可以用于队列和堆栈数据结构。
- defaultdict，即可返回默认值的字典类型。

应该说 Python 的数据类型已经相当丰富了。大多数应用并不需要再设计其他的数据结构。如果 Python 中的数据类型无法满足需求，则还可以使用 Collections 中的抽象基类构建自定义数据类型。只需要构建原生类型的子类并定制方法即可。

3.1.5　数据缓存

在许多场景中，二进制或者文本数据需要先缓存在 RAM 中的使用列表或字节数组。在传输应用中，后者更加适合做这方面的应用。

3.1.5.1　bytearray

```
>>> buf = bytearray(16)
>>> buf
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> for i in range(len(buf)):
...     buf[i] = 0x30 + i
```

```
...
>>> buf
bytearray(b'0123456789:;<=> ')
>>> type(buf[0])
<type 'int'>
```

在 bytearray 类型中的元素类型是整数，而不是字符，这更加接近物联网数据的本源。

3.1.5.2 StringIO/ByteIO

StringIO/ByteIO 实际是经过缓存的 I/O，分别对应字符串和二进制类型数据。原始的未经缓存的文件存取从磁盘直接读取文件（尤其是大型文件），这是一件耗费资源的事情，所以先将信息以 Stream 流式数据方式读取到内存中的缓存，然后程序从 StringIO/ByteIO 中读取，这样可以减少读取文件的次数。虽然操作系统现在提供的存取速度越来越快，硬件 IOPS 也越来越高，但是缓存 I/O 可以提供更加一致的性能。

StringIO/ByteIO 是纯 Python 实现，其 C 语言版本是 cStringIO/cByteIO，速度更快。StringIO 与 ByteIO 稍有区别，cStringIO 没有 len 和 pos 属性。在第 2 章中也提到过 StringIO/ByteIO，请注意使用上的各种限制。

```
>>> from io import BytesIO
>>> f = BytesIO(b'\x01\x23\x45\x67')
>>> f
<_io.BytesIO object at 0x00000000023E8F68>
>>> f.read()
'\x01#Eg'
>>> f.read()
''
```

为什么第二次 f.read 没有返回内容？因为程序已经读完了所有的缓存。ByteIO 不是 bytearray，这一点请体会其差别。

3.1.6 数据多路复用和解复用

在传输过程中，数据通道往往是共享的。数据多路复用（mux）和解复用（demux）是分享传输通道的重要手段。

3.1.6.1 字符合并

有时候我们手头处理的是字符的列表，这时可以利用 join 方法将其合并成字符串，如：

```
a = ['a','b','c','d']
s = ''.join(a)
print s # 'abcd'
```

如果 a 是整数，则略麻烦些。在 Python 2.7 中，可以采用列表推导式将整数转换成字符后再用 join 方法合并成字符串实现批量转换。Python 3 简单些，直接转换为 bytearray 类型。

```
>>> a = range(10)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b = [chr(i) for i in a]
>>> b
['\x00', '\x01', '\x02', '\x03', '\x04', '\x05', '\x06', '\x07', '\x08', '\t']
>>> repr(''.join(b))
"'\\x00\\x01\\x02\\x03\\x04\\x05\\x06\\x07\\x08\\t'"
>>> ba = bytearray(a)
>>> ba
bytearray(b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t')
```

常见的 MCU 内置 ADC 采集周围物理量，是多路复用的 ADC。一次轮询读取 ADC1，ADC2...ADC*n*。许多开发者就按照 ADC 采集时的多路复用数据经时间累积后原样上传。到了服务器端需要将多路 ADC 切片解复用，并按照每路 ADC 对应的物理量存放在缓存中或持久化到数据库中。在这些场景中，Python 可以使用的手段有 split 方法、列表切片、re 正则表达式等。本质上这些都是字符串的方法，只是 bytes/bytearray 继承了这些方法。

3.1.6.2　切片

采用切片来解决数据的多路解复用。利用序列类型的索引可以很容易地对部分或全部对象进行正向、反向切片。详情在前面已经有所涉及。

```
>>> b = 'abcdABCDwxyzWXYZ'
>>> c = tuple(b[i:i+4] for i in xrange(0,len(b),4))
>>> c
('abcd', 'ABCD', 'wxyz', 'WXYZ')
```

3.1.6.3　字符串切割

在物联网应用中，TCP 报文是流式数据，需要按照边界进行报文剪切，最常见的方法是根据 CR/LF 结束符组合进行报文剪切。也可能是通过某种报文帧结构的封装进行剪切。总之，需要对流入的字符串流或字节串流进行剪切。在 Python 中，按照特定字符串进行剪切很容易。

假设在二进制数据包帧结构中，0xFA 为包头，而 0xEE、0xFF 为包尾。报文中的最后一个报文没有包尾，以测试不同方法的结果。

```
stream = '\xFA\xAA123456\xEE\xFF\xFA\xBBa1b2c3d4\xEE\xFF\xFA\xCCQ!678#R$'
print stream.split('\xEE\xFF')

['\xfa\xaa123456', '\xfa\xbba1b2c3d4', '\xfa\xccQ!678#R$']
```

这样可以一次性地分解出三个报文帧结构。虽然 0xEE0xFF 被截取了，但是荷载数据已经

得到了正确解析。

3.1.6.4 正则表达式

同样针对上面的例子。如果了解正则表达式则可以更加“优雅”地解决该问题。导入 re 模块可以这样做：

```
>>> import re
>>> re.split("\xfa*\xee\xff", stream)
['\xfa\xaa123456', '\xfa\xbba1b2c3d4', '\xfa\xccQ!678#R$']
>>> re.split("\xfa*\xff", stream)
['\xfa\xaa123456\xee', '\xfa\xbba1b2c3d4\xee', '\xfa\xccQ!678#R$']
>>> re.split("\xfa*", stream)
['', '\xaa123456\xee\xff', '\xbba1b2c3d4\xee\xff', '\xccQ!678#R$']
```

后面两个表达式的切割结果完全不同，这体现了正则表达式的强大和灵活。不过正则表达式是一把“双刃剑”，二进制字符的正则匹配规则不同于人眼可阅读的字符串匹配，需要对协议完整性做一系列的完整测试，否则容易出错。开发者需要仔细阅读相关基础内容。

3.1.6.5 数据轴向转换

所谓轴向转换，指的是复用和解复用的另外一种理解。假设，前端传感器收集温度、湿度、振动的物理量。其排列是这样的：

```
time1,temp1,hum1,vibr1,time2,temp2,hum2,vibr2,time3,temp3,hum3,vibr3
```

但是我们在绘制其时序图表时，需要的是以下三种连续物联量：

```
time1,temp1,time2,temp2,time3,temp3
time1,hum1,time2,hum2,time3,hum3
time1,vibr1,time2,vibr2,time3,vibr3
```

我们可以使用列表推导式（list comprehension）来完成轴向转换。

```
>>> a = [[1,2,3],[4,5,6],[7,8,9]]
>>> b = [[row[col] for row in a] for col in range(len(a[0]))]
>>> b
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

还可以使用 zip 函数来实现。

```
>>> a = [[1,2,3],[4,5,6],[7,8,9]]
>>> b = zip(*a)
>>> b
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
>>> b = map(list,zip(*a))
>>> b
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

zip 是 Python 的一个内置函数，以可迭代的对象作为参数，将对象中对应的元素打包成 tuple（元组），然后返回由这些 tuple 组成的 list（列表）。若传入参数的长度不等，则返回 list 的长度和参数中长度最短的对象相同。利用*号操作符，可以将 list unzip（解压）。

map(list,zip(*a))涉及函数化编程，是将所有 tuple 转换成 list。这是一种函数化编程，与 C 语言相比，抽象程度可以更高。

轴向转换的应用场景除了数据的复用与解复用，还可以将其作为字模（font）的轴向转换。传统的 LCD/LED 点阵字模的转换涉及多个方向的转换：

- 上下颠倒；
- 左右反转（也是前后反转）；
- *X*/*Y* 轴转换。

上下和左右反转可以使用列表的正反切片方法。*X*/*Y* 轴转换可以使用 zip 来进行轴向转换。

我们再想想如何进行 3D 空间的 *XYZ* 轴向变换以及角速度的三轴转换？这会在许多物联网应用场合中用到。

3.1.7　数据序列化和反序列化

在数据传输时，需要在 Python 内部对象和传输码流之间进行转化，这被称为序列化和反序列化。有多种方式可以实现序列化与反序列化。

- 二进制数据：struct 模块，bytearray 类型的各类方法。
- 半结构化数据：XML 模块、JSON 模块。
- ASCII 编码：base64 模块、uuencode 模块。
- pickle/cPickle，不推荐。

3.1.7.1　struct 序列化二进制数据

Python 收发二进制数据时都进行类型转换，尤其是处理更加复杂的类型，如长整型、浮点数时都做类型转换，显得很烦琐。在 Python 中有个重要的 struct 包，专门用于二进制处理，其最小处理单元为字节。struct 这个名称和 C 语言中的 struct 有关联。可以参见表 3-3 了解 struct 包的格式字符与 C 语言数据类型的转换关系。

读者可能希望获得基于位的处理包。很抱歉，Python 里没有。据笔者所知，只有非常基础的 MCU（如 8051）才有位寻址；许多架构如 AVR/ARM，其最小寻址单位都是字节。在高级语言中（包括 C 语言），采用逻辑和移位计算可以很容易地获取位，代价是付出更多的计算资源。

表 3-3　struct 包中所用格式字符的意义

格式符号	C 语言数据类型	Python 数据类型	字节数
x	padding byte	None	1
c	char	长度为 1 的 string	1
b	signed char	integer	1
B	unsigned char	integer	1
?	boolean	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer 或 long	4
l	long	long	4
L	unsigned long	long	4
q	long long	long	8
Q	unsigned long long	long	8
f	float	float	4
d	double	float	8
s	char[]	string	1
p	char[]	string	1
P	void *	long	native

为了与 C 语言的结构体交换数据，需要考虑 C/C++编译器的字节对齐。struct 可以利用格式化字符串的第一个字符来定义字节对齐方式，其含义可参见表 3-4。

表 3-4　struct 包中的对齐方式

格式化字符	字节顺序	尺　寸	对　齐
@	native	native	凑齐 4 字节
=	native	标准	原字节数
<	little-endian（小端）	标准	原字节数
>	big-endian（大端）	标准	原字节数
!	network（=大端）	标准	原字节数

假设我们需要从通信端口（大端模式）获取数据的报文结构体如下：

```
struct FrameDataPoint
{
```

```
    unsigned int header;
    unsigned int datalen;
    unsigned char version;
    unsigned char snr[6];
    unsigned char samplerate;
    unsigned char pressure;
    unsigned char flow;
    unsigned char leak;
    unsigned char frequency;
    unsigned int crc16;
    unsigned int trailer;
}
```

假设收到的字节流数据符合以上数据结构，则可以使用如下代码进行解析：

```
import struct
(header, datalen, version, snr, samplerate, pressure, flow, leak, freq, crc, trailer)
 = struct.unpack(">HHB6BBBBBBHH", s)
```

发送端可以采用 struct.pack 进行编码。

以上数据类型转换、编解码方法，适用于大多数二进制数据交换：如文件操作、串口和网络操作等。

需要注意的是 struct.unpack 的返回结果是元组，是不可变动类型。在某些情况下，解包之后，还需要对这些类型进行转换。需要先转换到列表类型，才可以进行处理。

```
>>> import struct, json
>>> s = 'hello world!'
>>> sz = len(s)
>>> pat = '!%dB'%(sz)
>>> t = struct.unpack(pat,s)
>>> t
(104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100, 33)
>>> l = list(t)
>>> l
[104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100, 33]
>>> l = map(lambda x:x/5.0, l)
>>> l
[20.8, 20.2, 21.6, 21.6, 22.2, 6.4, 23.8, 22.2, 22.8, 21.6, 20.0, 6.6]
>>> json.dumps(t)
'[104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100, 33]'
>>> json.dumps(l)
'[20.8, 20.2, 21.6, 21.6, 22.2, 6.4, 23.8, 22.2, 22.8, 21.6, 20.0, 6.6]'
```

在上例中，使用 unpack 方法导出 tuple，然后利用 lambda 和 map 方法在一条语句中完成列表容器中每个条目除以 5.0 浮点数的操作。最后 json 输入参数无论是 list 还是 tuple 类型，输出 JSON 格式是一致的。所以并不需要再额外转回 tuple。

3.1.7.2 bytes/bytearray 方法

bytes/bytearray 支持常见的序列类型操作。此外，其还继承了字节类对象的操作。

fromhex 方法可以从一个 HEX 字符串返回一个 byte/bytearray 对象。

```
>>> bytearray.fromhex('30406032')
bytearray(b'0@`2')
>>> bytes.fromhex('30406032')
b'0@`2'
```

此外，bytes/bytearray/str 的构建函数可以作为彼此类型转换的途径。但是在 Python 2/3 中存在一些差异。推荐在 Python 3 中优先采用 bytearray/bytes 为基础来作为物联网的数据类型。

```
>>> b = b'Hello'
>>> s = 'World'
>>> bytes(s,encoding="utf8")
>>> str(b,encoding="utf8")
>>> str(b)
"b'Hello'"
>>> bytes(s)
>>> str.encode(s)
b'World'
>>> bytes.decode(b)
'Hello'
>>>
```

上述例子在 Python 2.7.11、Python 3.4 和 MicroPython 中得到的结果和实现均不相同。使用时需谨慎，最好事前在 REPL 中测试一番。

3.1.7.3 半结构化数据序列化

XML、XHTML、SGML、JSON 属于半结构化和准结构化数据，是网络中常见的数据格式。

1．XML 模块

XML（eXtensible Markup Language，可扩展标记语言）被设计用来传输和存储数据，其已经日趋成为当前许多新生技术的核心，在不同的领域都有着不同的应用。它是 Web 发展到一定阶段的必然产物，其既具有 SGML 的核心特征，又有着 HTML 的简单特性，还具有明确和结构良好等特征。

Python 解析 XML 有四种方法：

- xml.dom.*模块，W3C DOM API 的实现。很适合处理 DOM 模型，同样可以用于产生 XML 文件。
- xml.sax.*模块，SAX API 的实现。SAX 是一个基于事件的 API，可以不用完全加载进内存，以流式数据形式处理庞大数量的文档。SAM 以牺牲访问便捷性，换取了高速和

资源节省。

- xml.etree.ElementTree 模块，简称 ET。ET 提供了轻量级的 Python API。ET 不仅比 DOM 快，其还有许多其他 API。ET.iterparse 也提供了类似 SAX 的处理方式，也无须加载全部文档。ET 的性能和 SAX 差不多，但是 API 效率更高，使用更方便。
- xml.parsers.expat，一个面向流的解析器。首先注册解析器回调（或 handler）功能，然后开始搜索文档。当解析器识别后会调用该部分相应的处理程序。XML 文件被输送到解析器时，会被分隔成多个片断，并分段装到内存中。因此，expat 可以解析那些巨大的文件。

xml.etree.ElementTree 还有对应的 C 语言版本：cElementTree。其速度更快，内存消耗更少。xml 模块是 Python 标准库之一。

我们先介绍如何利用 xml.dom.minidom 产生 XML 文件。

XML_DOM_gen.py：

```
import xml.dom.minidom as Dom

if __name__ == "__main__":
    doc = Dom.Document()
    root_node = doc.createElement("family")
    root_node.setAttribute("name", "liu")
    root_node.setAttribute("url", "http://www.ennovation.org")
    doc.appendChild(root_node)

    kid_node = doc.createElement("kid")

    kid_name_node = doc.createElement("name")
    kid_name_value = doc.createTextNode("kirin")
    kid_name_node.appendChild(kid_name_value)
    kid_node.appendChild(kid_name_node)

    kid_age_node = doc.createElement("age")
    kid_age_value = doc.createTextNode("5")
    kid_age_node.appendChild(kid_age_value)
    kid_node.appendChild(kid_age_node)

    root_node.appendChild(kid_node)

    f = open("family.xml", "w")
    f.write(doc.toprettyxml(indent = "\t", newl = "\n", encoding = "utf-8"))
    f.close()
```

family.xml：

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<family name="liu" url="http://www.ennovation.org">
  <kid>
    <name>kirin</name>
    <age>5</age>
  </kid>
</family>
```

单纯利用 Python 程序产生 XML 文件，有些烦琐。我们可以采用模板技术来产生 XML。这可以参考 Python Web 框架中的 Jinja2 或者其他模板技术，而不一定采用 DOM 包来产生。

2．lxml 包

除了 XML，开发者还可以在工程中使用 lxml。lxml 是 C 运行库 libxml2，libxslt 的 Python 的封装库。它兼顾速度、Python 处理 XML 的便捷性，并与 ElementTree（ET）模块兼容。lxml 支持 XML、XHTML、XPath 和 XSLT、Schema 等。可在前面 family.xml 的 family 根节点下挂接更多 kid，并利用 lxmlparser.py 来解析，如下所示。

lxmlparser.py：

```
from lxml import etree

class findTarget(object):
    def __init__(self):
        self.kid = []

    def start(self, tag, attrib):
        self.is_name = True if tag == "name" else False
        self.is_age = True if tag == "age" else False

    def end(self, tag):
        pass

    def data(self, data):
        if self.is_name:
            self.kid.append(data)
            self.is_name = False
        if self.is_age:
            pass

    def close(self):
        return self.kid

parser = etree.XMLParser(target = findTarget())

doc = etree.parse('family.xml', parser)
print doc
```

运行结果如下：

```
[u'kirin', u'future']
```

3．性能比较

根据客户的需求，需要使用 XML 格式文档作为服务器与 PC 客户端之间的信息交换。笔者最初采用了 xml.dom.minidom 来产生 XML 文件，但却发现其耗时非常长。20000 个数据点需要 90 秒。数据点从数据库查询、构建 XML 到传输，过程中有多个环节。在做了交叉测试后，发现传输不是瓶颈；数据库查询才是最大瓶颈，需要优化。此外，XML 创建也需要进行优化。所以，在此针对 XML 创建这一环节做了一些测试，并分别对 minidom、lxml 和第三方的 BytesIO SteamingXmlWriter 做了测试。

benchmark.py：

```
#!/usr/bin/env python

# for streamingXmlWriter

#from __future__ import print_function
from io import BytesIO
from collections import OrderedDict
import streamingxmlwriter

# for minidom
import xml.dom.minidom as Dom

# for lxml
from lxml import etree as ET

import datetime
import time
import random

d_data = []
dom_data = None
tree = None
stream = BytesIO()

def genData():
        global d_data
        for i in xrange(10000):
            #print(i)
            r = random.randint(1,100)
            snr = random.randint(1,100000)
            date = random.randint(1,100000)
            tag1 = random.randint(0,255)
            tag2 = random.randint(0,255)
            tag3 = random.randint(0,255)
```

```
            tag4 = random.randint(0,255)
            tag5 = random.randint(0,255)
            tag6 = random.randint(0,255)
            d = dict(snr=snr, date=date, tag1=tag1, tag2=tag2, tag3=tag3, tag4=tag4,
tag5=tag5, tag6=tag6)
            d_data.append(d)

def writerByDom():
        global d_data, dom_data
        doc = Dom.Document()
        root_node = doc.createElement("logging")
        root_node.setAttribute("name", "allankliu")
        root_node.setAttribute("url", "http://www.ennovation.org")
        doc.appendChild(root_node)

        for i in d_data:
            d_node = doc.createElement("device")

            d_snr_node = doc.createElement("snr")
            d_snr_value = doc.createTextNode(str(i['snr']))
            d_snr_node.appendChild(d_snr_value)
            d_node.appendChild(d_snr_node)

            d_date_node = doc.createElement("date")
            d_date_value = doc.createTextNode(str(i['date']))
            d_date_node.appendChild(d_date_value)
            d_node.appendChild(d_date_node)

            d_tag1_node = doc.createElement("tag1")
            d_tag1_value = doc.createTextNode(str(i['tag1']))
            d_tag1_node.appendChild(d_tag1_value)
            d_node.appendChild(d_tag1_node)

            d_tag2_node = doc.createElement("tag2")
            d_tag2_value = doc.createTextNode(str(i['tag2']))
            d_tag2_node.appendChild(d_tag2_value)
            d_node.appendChild(d_tag2_node)

            d_tag3_node = doc.createElement("tag3")
            d_tag3_value = doc.createTextNode(str(i['tag3']))
            d_tag3_node.appendChild(d_tag3_value)
            d_node.appendChild(d_tag3_node)

            d_tag4_node = doc.createElement("tag4")
            d_tag4_value = doc.createTextNode(str(i['tag4']))
            d_tag4_node.appendChild(d_tag4_value)
            d_node.appendChild(d_tag4_node)
```

```
            d_tag5_node = doc.createElement("tag5")
            d_tag5_value = doc.createTextNode(str(i['tag5']))
            d_tag5_node.appendChild(d_tag5_value)
            d_node.appendChild(d_tag5_node)

            d_tag6_node = doc.createElement("tag6")
            d_tag6_value = doc.createTextNode(str(i['tag6']))
            d_tag6_node.appendChild(d_tag6_value)
            d_node.appendChild(d_tag6_node)

            root_node.appendChild(d_node)
        dom_data = root_node

def domToFile():
        global dom_data
        f = open('dom.xml','w')
        f.write(dom_data.toprettyxml(indent = "\t", newl = "\n", encoding = "utf-8"))
        f.close()

def writerByLxml():
        global d_data, tree

        logging = ET.Element('logging')
        logging.attrib['name'] = 'allankliu'
        logging.attrib['url']='http://ennovation.org'

        for i in d_data:
            device = ET.SubElement(logging, 'device')
            snr = ET.SubElement(device, "snr")
            snr.text = str(i['snr'])
            date = ET.SubElement(device, "date")
            date.text = str(i['date'])

            tag1 = ET.SubElement(device, "tag1")
            tag1.text = str(i['tag1'])
            tag2 = ET.SubElement(device, "tag2")
            tag2.text = str(i['tag2'])
            tag3 = ET.SubElement(device, "tag3")
            tag3.text = str(i['tag3'])
            tag4 = ET.SubElement(device, "tag4")
            tag4.text = str(i['tag4'])
            tag5 = ET.SubElement(device, "tag5")
            tag5.text = str(i['tag5'])
            tag6 = ET.SubElement(device, "tag6")
            tag6.text = str(i['tag6'])

        tree = ET.ElementTree(logging)
```

```
def treeToFile():
        global tree
        tree.write("lxml.xml", pretty_print=True, xml_declaration=True, \
        encoding='utf-8')

def writerByStreamXml():
        global d_data, stream
        #stream = BytesIO()
        with streamingxmlwriter.from_stream(stream) as writer:
            writer.start_namespace('myns', 'http://mynamespace.org/')
            with writer.element('logging', \
            {'name': 'allankliu','url':'http://ennovation.org/'}):
                for i in d_data:
                    with writer.element('device'):
                        with writer.element('snr'):
                            writer.characters(str(i['snr']))
                        with writer.element('date'):
                            writer.characters(str(i['date']))
                        with writer.element('tag1'):
                            writer.characters(str(i['tag1']))
                        with writer.element('tag2'):
                            writer.characters(str(i['tag2']))
                        with writer.element('tag3'):
                            writer.characters(str(i['tag3']))
                        with writer.element('tag4'):
                            writer.characters(str(i['tag4']))
                        with writer.element('tag5'):
                            writer.characters(str(i['tag5']))
                        with writer.element('tag6'):
                            writer.characters(str(i['tag6']))

def streaming2File():
        global stream
        file = open('streaming.xml','wb')
        file.write(stream.getvalue())
        file.close()

def performance(fname):
        print fname.__name__
        start = datetime.datetime.now()
        r = fname()
        delta = datetime.datetime.now() - start
        print(delta)

def main():
        performance(genData)
        performance(writerByDom)
        performance(domToFile)
```

```
        performance(writerByLxml)
        performance(treeToFile)
        performance(writerByStreamXml)
        performance(streaming2File)

if __name__=='__main__':
        main()
```

运行结果如下：

```
> benchmark.py
genData
0:00:00.593000
writerByDom
0:00:04.264000
domToFile
0:00:04.314000
writerByLxml
0:00:00.390000
treeToFile
0:00:00.129000
writerByStreamXml
0:00:15.460000
streaming2File
0:00:00.006000
```

采用随机数创建 Python 字典用了 0.6 秒的时间；minidom 方式创建 XML 和写入文件分别用了 4.26/ 4.31 秒的时间；lxml 方式分别用了 0.39/0.13 秒的时间；而 streamingAPI 分别用了 15.46/0.006 秒的时间。综合下来，lxml 用于生成 XML 的速度最快。让笔者比较奇怪的是 streamingAPI 创建 XML 为何比 minidom 还要慢？

以上代码中的 XML tag 生成代码可以进一步简化，这个问题由读者自行思考。

4．JSON 模块

JSON 是现有 REST API 的主流形式。JSON 的编码和解码在 Python 中可以通过 json 包完成。JSON 与 Python 数据类型的转换关系表如表 3-5 所示。

表 3-5　Python/JSON 数据类型转换关系表

Python	JSON
dict	object
list，tuple	array
str，unicode	string
int，long，float	number
True	true

续表

Python	JSON
False	false
None	Null

REPL 测试：

```
>>> import json
>>> raw = [1,2,3,"xyz",(1,2,3),[4,5,6],{"name":"allankliu","city":"sha"}]
>>> ej = json.dumps(raw)
>>> raw
[1, 2, 3, 'xyz', (1, 2, 3), [4, 5, 6], {'city': 'sha', 'name': 'allankliu'}]
>>> ej
'[1, 2, 3, "xyz", [1, 2, 3], [4, 5, 6], {"city": "sha", "name": "allankliu"}]'
>>> print json.dumps(raw, indent=4)
[
    1,
    2,
    3,
    "xyz",
    [
        1,
        2,
        3
    ],
    [
        4,
        5,
        6
    ],
    {
        "city": "sha",
        "name": "allankliu"
    }
]
>>> ej.decode()
u'[1, 2, 3, "xyz", [1, 2, 3], [4, 5, 6], {"city": "sha", "name": "allankliu"}]'
```

以上代码很简略地介绍了 JSON 的编码和解码过程，JSON 的更多方法请参考文档。

3.1.7.4 字符串序列化

出于各种原因，二进制数据有时候无法以原始数据形式传输，而需要转化为 ASCII 字符串（7 位 ASCII 字符串）进行传输。电子邮件、Web 传输，以及 M2M 模块通信、短消息收发等，都采用了各种形式的 ASCII 字符串协议。该协议包括 uuencode 和 base64 等。

Python 处理 base64 有两种软件包：binascii 和 base64。相对来说，base64 更加符合 RFC3548

的规定。

```
import binascii
import base64

text = 'hello'
b0 = binascii.b2a_base64(text)
b1 = base64.b64encode(text)
print b0 # 'aGVsbGB=\n'
print b1 # 'aGVsbGB='
print binascii.a2b_base64(b0) # 'hello'
print base64.b64decode(b1) # 'hello'
```

注意：binascii.b2a_base64 方法返回值的最后附加了一个'\n'。不要在这种“小坑”里浪费时间。

3.1.7.5　对象的序列化

Python 的各种对象可以在内存中。但其只有序列化之后才可以进行存储和交换。一般意义上的序列化，可以采用 pickle 来实现。采用 pickle 序列化 Python 对象后，可以保存在硬盘中。这样系统断电后还可以重新加载，并反序列化为 Python 对象。pickle 和 cPickle 的区别在于 cPickle 采用 C 来实现，比 pickle 要快。但是反序列化后的对象是 Python 对象，这在使用上有一定的安全风险。

如果要通过套接字与外界进程进行交换，JSON 会是一种标准方法。可以将 Python 序列化为 JSON 后，保存并与外界进行信息交换。由于 JSON 并非 Python 内置对象，所以可以进行额外的安全检查。

3.1.8　数据压缩和解压缩

出于降低系统运行成本、节省资源的目的，数据压缩和解压缩在存储和传输中得到了大量应用。其中文本信息的压缩比相当大，包括 XML/JSON/HTML 都可以进行压缩。在许多 Web 服务器中，启动 gzip 压缩可以节约带宽支出，加快传输速度。一些海量数据在归档时也需要采用压缩归档。

Python zipfile 是常见的压缩/解压缩包，内有 ZipFile 和 ZipInfo 两个模块。

readzip.py：

```
import zipfile
z = zipfile.ZipFile(filename,"r")
for fn in z.namelist():
    print fn

for i in z.infolist():
    print fn.file_size, fn.header_offset
```

```
print z.read(z.namelist()[0])
```

writezip.py：

```
import zipfile
z = zipfile.ZipFile(filename,"r")
z.write("temp.txt")
z.close()
```

注意，某些黑客软件利用 zipfile 中 extractall(path =,member=,pwd=)方法的密码参数来实现暴力破解密码。原理很简单，就是利用密码字典中的密码一个个去试。所以，zip 密码要使用强密码。

3.1.9 数据加密

在互联网上传输，必须有个合适的安全通道。加密算法可以用于认证、授权、加密算数和存储。实际上这方面的 Python 模块和包特别多：

- 标准库 crypto，支持对称 GPG 文件加密和解密、AES256/SHA256 算法等。
- pyopenssl，OpenSSL 的 Python 封装库。
- hashlib，专门收集哈希算法如 MD5/SHA1/SHA224/SHA256/SHA384/SHA512 的软件包。
- pycrypto，收集了安全哈希算法如 SHA256/RIPEMD160，以及多种加密算法（AES/DES/RSA/ElGamal 等）。该软件包架构有利于不断添加新模块。另外，还内置了一个随机数发生器模块。

其中，pycrypto 很适合用于各类加密和哈希算法的安全算法的设计、验证与使用。使用 pip 安装 pycrypto：

```
pip install pycrypto
```

REPL 测试：

```
>>> from Crypto.Hash import SHA256
>>> hash = SHA256.new()
>>> hash.update('hello')
>>> hash.digest()
',\xf2M\xba_\xb0\xa3\x0e&\xe8;*\xc5\xb9\xe2\x9e\x1b\x16\x1e\\\x1f\xa7B^s\x043b\x93\
x8b\x98$'
>>> from Crypto.Cipher import AES
>>> obj = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
>>> message = "The answer is no"
>>> ciphertext = obj.encrypt(message)
>>> ciphertext
'\xd6\x83\x8dd!VT\x92\xaa`A\x05\xe0\x9b\x8b\xf1'
```

```
>>> obj2 = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
>>> obj2.decrypt(ciphertext)
'The answer is no'
>>> from Crypto import Random
>>> rnd = Random.new()
>>> rnd.read(16)
'\x16\xd7\x08M\xc8b\xc7\t\xe0X\xa9RCPYJ'
```

各类哈希函数中的 update 方法会保存使用过的字符串，所以新字符串计算哈希时需要重新创建实例。

在安全领域，如何充分利用哈希算法，读者需要研究一下各类对称和非对称算法。Python 应用中采用以上软件包可实现系统安全的快速搭建和验证。

3.1.10　数据传输

在 Python 的标准库里，socket 是 TCP/IP 传输的底层通信包。基于 socket，标准库里还有许多服务器和客户端的实现，包括 FTP、SMTP/IMAP、Telnet、NNTP、HTTP Web 服务器/客户端、urllib/urllib2，以及基于 HTTP 的 JSON/XML、RPC 等。此外，还有数据加密 TLS 等通信包。

许多物联网通信框架底层都是基于 socket 开发的。但是开发者如果使用标准库的设计，往往只能够实现较为简单的设计，还有太多细节需要重新实现。所以笔者推荐使用 Twisted 进行 UDP/TCP 套接字客户端和服务器端开发，使用更为流行的 Django、Flask、Tornado 和 Cyclone 框架来实现更为成熟完整的 Web 服务器设计。

标准库中的一些客户端例子很适合作为测试目的或者客户端应用的基础，这时候使用 Twisted 或 Scrapy 爬虫太重，标准库正好用得上。但如果需要做压力测试，还是必须使用高并发的异步或协程框架来制作压力测试专用客户端。

3.1.11　数据后处理

数据后处理，本质上是数据的反序列化、切割、清晰和转换等。利用前面的序列化、类型转换、struct 进行，以及匿名函数、map、reduce、filted、sorted、列表推导式、切片切块等 Python 语言高级特性，原始数据可以快速得到批量处理。

3.1.12　数据持久化

在物联网应用中，数据持久化的选项非常多。pickle/cpickle、sqlite3、XML、JSON、dbm，以及大量第三方库都使得 Python 能够支持大多数数据持久层技术。

虽然许多数据库后端都可以使用原生 API 访问，但是出于系统移植的其他需求，越来越多的人使用 ORM（Object Relational Mapping），即将 Python 中的对象映射到数据库字段的方式。

这样，我们在具体操作业务对象时，就不需要再去和复杂的 SQL 语句打交道，只需简单地操作对象的属性和方法。流行的 Python ORM 软件包如下：

- SQLAlchemy，使用最广泛的 Python ORM，作者是 Michael Bayer。
- SQLObject，ActiveRecord 模式，提供基于 Python 对象的查询语言，这使得 SQL 更加抽象。
- Storm，Ubuntu Canonical 开发的 ORM。
- Django ORM，与 Django 配合使用的 ORM。
- peewee，小巧简单的 Python ORM。

相比其他的 ORM，SQLAlchemy 是学习周期较长，而且专注于工作单元的前沿概念。但企业级 API 使得代码有健壮性和适应性，设计灵活，支持复杂查询。

几乎所有存储后端都有对应的 Python 封装，但需要注意同步和异步框架的接口选择区别。

3.1.13 数据交换

除了 JSON 和 XML 之外，另一种流行的数据交换方式是 CSV。该文件以逗号和引号作为字段分隔符，其实际上是一种二维表格，得到了大多数数据表软件（如 Excel）和数据库（如 MySQL）的支持，可以归类为结构化数据。CSV 文件在批量数据迁移、处理和分析中得到了大量使用。

sample.csv：

```
name,age,addr,tel
allan,30,shanghai,123456
george,40,shanghai,8765431
michael,22,wuhan,12909834
rebecca,30,hongkong,12190093
```

plaincsv.py：

```
for line in open("sample.csv","r"):
    name,age,addr,tel = line.split(',')
    print name, age, addr, tel
```

democsv.py：

```
import csv
reader = csv.reader(open("sample.csv"))
for name, age, addr, tel in reader:
    print name, age, addr, tel
```

一般来说，Python 语言本身的功能足够应付简单的 CSV 文件，不同 CSV 文件有些细微差异：如分隔符、双引号、转义符等定义有所不同。这些细节可以通过参数设置交给 csv 模块解决。

3.2　HOWTO：函数式编程

函数式编程（Funtional Programming）的“函数”不是编程语言中的函数，而是数学意义上的抽象表达式。

函数式编程是一种抽象层次很高的编程范式。我们常见的编程范式有命令式编程、函数式编程和逻辑式编程。OOP 也是一种命令式编程。函数式编程应对了当今世界上日益增长的并行性编程和元数据编程趋势。

函数式编程的特点如下：

- 闭包和高阶函数（enclosure & high-order function）；
- 惰性计算（lazy evaluation）；
- 递归（recursion）；
- 引用透明性；
- 副作用（side-effect）。

函数式编程的典型代表语言为 Lisp 和 Haskell，其他还有 Scala/Clojure 和 Erlang。许多语言如 C/C++/Java/Python 可以通过特定方式部分支持函数式编程。详情可参阅本章延伸阅读部分中的文章。我们来看看 Python 中的函数式编程。Python 的 functools 模块中包含了一些函数式编程的方法。

Python 中的函数式编程特点如下：

- 不是纯函数式编程，允许变量存在；
- 支持高阶函数，可传入函数做变量；
- 支持闭包，可返回函数；
- 有限度地支持匿名函数，必须在一行内完成。

3.2.1　高阶函数

何谓高阶函数？首先，变量可以指向对象，也可以指向函数。即函数名可以赋值给一个变量，使用这个变量可以完成函数功能。调用该变量的函数就被称为高阶函数。

```
>>> f = abs
>>> f(-100)
100
```

其实，在 C/C++中也有类似的情况，尤其是底层函数调用高层函数，即在使用回调（callback）函数过程中，注册回调函数时，就是将函数名作为参数进行注册的。而后在回调发生时，利用函数指针来实现调用。

```
void gsmRing(){
```

```
    // user logic when incoming call rings.
}

gsm.registerCallback(gsmRing,GSM::CbRING);

void registerCallback(void (*fptr)(void), CallbackType type) {
    if (fptr) {
        _callbacks[type].attach(fptr);
    }
}
```

回到 Python 中，如果把函数 A 名称作为函数 B 的参数，那么函数 B 就是 A 的高阶函数。Python 内置了许多高阶函数，如 map/reduce/filter/sorted 等函数。

3.2.2 map 函数

map 函数有两个参数，一个是函数名，另一个是序列类型对象。map 将函数名依次作用于序列的每个元素，并把结果作为新的迭代器返回。

```
>>> def f(x):
...     return x/5.0
...
>>> map(f,[1,3,5,7,9,11,13,15])
[0.2, 0.6, 1.0, 1.4, 1.8, 2.2, 2.6, 3.0]
```

虽然我们可以使用 for 循环实现这个目的，但是 map 函数帮助我们把运算规则抽象化了。在 f 函数中，除了简单计算用于物理量的缩放，也可以做许多类型转换和其他操作与计算。

3.2.3 reduce 函数

reduce 函数参数也是函数名和序列类型对象。即把一个函数作用在一个序列上，这个函数必须接受两个参数，reduce 函数把结果继续和序列的下一个元素做累计计算。

```
reduce(f, [x1,x2,x3,x4] = f(f(f(x1,x2),x3),x4)
```

换而言之，reduce 函数将一个需要反复迭代的算法简化为迭代函数与数据序列的表达式。map 函数/reduce 函数可以简化不少编程工作量。

3.2.4 filter 函数

filter 函数参数包括一个函数名和序列，并过滤该序列。filter 函数把函数逐个作用于序列中的每个元素，根据返回的 True/False 决定是保留还是丢弃元素。所以，filter()返回序列是原序列的过滤结果。

```
>>> def isOdd(x):
...     return x % 2 == 1
...
>>> filter(isOdd, range(10))
[1, 3, 5, 7, 9]
```

3.2.5　sorted 函数

排序也是各类程序中的常见需求，包括数值类的排序、文字排序，甚至是类型排序和数据统计等。

```
>>> sorted([2321,234324,324324,3248372,23432423,324234])
[2321, 234324, 324234, 324324, 3248372, 23432423]

>>> sorted(['all','English','Chinese','French'], key=str.lower)
['all', 'Chinese', 'English', 'French']
```

在上例中，sorted 函数中采用 key=str.lower 进行排序；也可以是自定义的函数名，该函数中可以实现用户自己的比较算法。比较算法通过 sorted 高阶函数抽象化。

3.2.6　返回函数

高阶函数可以接受函数名作为参数，也可以返回函数名作为结果。

```
>>> def mult(*args):
...    def prod():
...       r = 1
...       for m in args:
...          r = r*m
...       return r
...    return prod
...
>>> f = mult(1,2,3,4)
>>> f()
24
```

在上例中，函数 prod 作为 mult 的返回值。在给 f 函数赋值时没有发生任何计算，在 f 函数被调用时才进行 prod 计算。这就是所谓的 lazy evaluation——惰性计算。这是一个计算机编程概念，它的目的是最小化计算机要做的工作。惰性计算的相反是热情计算，也叫作严格计算，它是大多数编程语言现有的普通计算方式。

惰性计算有两个相关而又有区别的含意，可以表示为“延迟计算”和“短路求值”。本节的例子仅专注前者，后者请参见 Python 对象真值计算操作。除了可以提升性能，惰性计算还可以在有限的资源中构造一个无限的数据类型。

3.2.7 闭包

还是采用 3.2.6 节中的代码例子。mult 函数内的 prod 函数可以应用 mult 的参数 args 和局部变量 r，但 mult 将 prod 作为返回值时，参数和变量也保持在返回函数中，而没有被释放。这种返回内部函数及其上下文（即外部函数的参数和变量）的结构叫作闭包。说起闭包，笔者就想起了 JavaScript。闭包是 JavaScript 之所以强大的原因之一，也是大家学习的难点之一。

```
>>> f = mult(1,2,3,4)
>>> repr(f)
'<function prod at 0x00000000022F69E8>'
>>> fn = mult(1,2,3,4)
>>> repr(fn)
'<function prod at 0x00000000022F6A58>'
```

这说明 f()和 fn()尽管均调用 mult(1,2,3,4)，但却是不同的对象。如果将这种闭包模型针对多个网络连接的函数，那么这意味着每个函数都可以一直保持连接，直到内部函数运行结束为止。

3.2.8 匿名函数

与 JavaScript 将匿名函数作为常规功能不同，Python 的匿名函数是有限度的。在许多时候，将函数名作为参数传入高阶函数时，可以不必单独定义函数，而可以利用匿名函数在一行语句中完成所有操作。

在之前的 map 函数例子中，可以利用以下匿名函数完成：

```
>>> map(lambda x:x/5.0, [1,3,5,7,9,11,13,15])
[0.2, 0.6, 1.0, 1.4, 1.8, 2.2, 2.6, 3.0]
```

在物联网应用中，经常需要对所采集的原始数据（Byte/Word）进行范围内缩放（scale），这种方式可以将整个的一个序列物理量进行快速缩放，而无须迭代和定义函数。

匿名函数也是函数对象，可以将匿名函数赋值给变量，利用变量来调用匿名函数。

```
>>> f = lambda x:pow(x,3)
>>> f
<function <lambda> at 0x00000000026F2D68>
>>> f(2)
8
```

lambda 在仅需要一段可执行代码的情况下，带来了更加简洁的代码结构。在嵌入式 MicroPython 中，可以用 lambda 函数实现一个简单的中断服务程序(ISR)。详情参见 6.6.10 节。

3.2.9　装饰器

装饰器（decorator）是 Python 最常见的“语法糖”之一。许多 Python 网络编程框架中大量使用了各种装饰器，以简化代码。decorator 在面向对象编程中被称为装饰模式。Python 采用@，从语法上支持了装饰模式编程。

```
>>> f = datetime.datetime.now
>>> f()
datetime.datetime(2016, 7, 1, 18, 48, 3, 145000)
>>> f = datetime.datetime.now
>>> f()
datetime.datetime(2016, 7, 1, 18, 48, 20, 285000)
>>> f.__name__
'now'
>>> def log(func):
...     def wrapper(*args, **kw):
...        print func.__name__ + ":"
...        return func(*args, **kw)
...     return wrapper
...
>>> @log
... def randomfunc():
...     print "Random figure"
...
>>> randomfunc()
randomfunc:
Random figure
>>>
```

log 函数是 func 函数的高阶函数，其参数全部交给 wrapper()。wrapper 用于显示 func 的函数名。@log 函数放置在函数定义前，等价于 log(randomfunc)。由于 log()可能是许多函数都需要用到的，所以这样做可以节省不少冗余代码。

如果 log 函数中还需要其他参数，比如 log('debug')、log('info')以区分不同的调试信息，则会更加复杂一些。对于读者来说，比较简单的做法是记住 functools 的用法。

不带参数：

```
import functools

def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print func.__name__
        return func(*args, **kw)
    return wrapper
```

带参数（三层嵌套）：

```
import functools

def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print '%s %s():' % (text, func.__name__)
            return func(*args, **kw)
        return wrapper
    return decorator
```

Python 装饰器

装饰器往往与面向切面编程有密切关联。

大家都熟悉面向过程（Procedure Oriented）和面向对象（Object Oriented）编程。但是有些场景需要面向切面编程（AOP，Aspect Oriented Programming），如插入日志、性能测试、事务处理等。面向切面作为一种著名的设计模式，可以用来抽离大量函数中与函数功能无关的重复代码并继续重用。

装饰器在面向切面编程中为现有对象增加了额外功能。在 Web 编程中，每个 URL Handler 函数中往往都需要做如下重复的工作。

（1）数据库连接：当前数据库是否为之后的数据库访问准备好。

（2）链接等待：由于数据库可能是慢查询，必要时需要交出控制权。

（3）参数过滤：HTTP 请求是否携带 cookie，参数是否合法。

（4）Session 检查：每个客户端是否在数据库或文件系统中存在对应的 Session。

（5）权限和访问控制：依据 RBAC（即角色访问控制）/ACL（访问控制列表）的设计方式，通过查询数据库或缓存中的角色定义与控制表单来判断连接的客户端是否有权限访问对应的资源。

（6）网页渲染：根据角色和权限，来打开和关闭视图中的某些按钮操作。

（7）出错处理：根据不同出错情况转到 HTTP403/404/50X 等网页。

（8）其他操作：日志记录、事件触发等。

这些重复的工作固然可以通过剪贴代码来做，但重复代码的可维护性很差，同时多个复合控制逻辑混合在一起，调试与维护也容易出错。这时候装饰器绝非可有可无的“装饰”，而是必须掌握的知识。所以，在主流 Web 框架中，都会有几个常见的装饰器。

在 Cyclone 中往往使用以下装饰器。

```
@cyclone.web.authenticated
@defer.inlineCallbacks
@storage.DatabaseSafe
```

这三个装饰器分别来自 Cyclone、Twisted 和项目本身代码。路径分别为：

- Python2.7\Lib\site-dist\cyclone\web.py，验证当前用户是否被认证过；
- Python2.7\Lib\site-dist\twisted\internet\defer.py，当前 Handler 有异步 Defer 回调需要处理；
- cyclone_project\storage.py，验证当前数据库是否连接。

这三个装饰器一下子节省了大量的代码和复杂的控制逻辑。需要注意的是，装饰器的引用顺序很重要。

web.py：

```
def authenticated(method):
    """Decorate methods with this to require that the user be logged in."""
    @functools.wraps(method)
    def wrapper(self, *args, **kwargs):
        if not self.current_user:
            if self.request.method in ("GET", "HEAD"):
                url = self.get_login_url()
                if " " not in url:
                    if urlparse.urlsplit(url).scheme:
                        # if login url is absolute, make next absolute too
                        next_url = self.request.full_url()
                    else:
                        next_url = self.request.uri
                    url = "%s %s" % (url,\
                                     urllib.urlencode(dict(next=next_url)))
                return self.redirect(url)
            raise HTTPError(403)
        return method(self, *args, **kwargs)
    return wrapper
```

defer.py：

```
def inlineCallbacks(f):
    @wraps(f)
    def unwindGenerator(*args, **kwargs):
        try:
            gen = f(*args, **kwargs)
        except _DefGen_Return:
            raise TypeError(\
                "inlineCallbacks requires %r to produce a generator; instead"\
                "caught returnValue being used in a non-generator" % (f,))
        if not isinstance(gen, types.GeneratorType):
            raise TypeError(\
                "inlineCallbacks requires %r to produce a generator; "\
                "instead got %r" % (f, gen))
        return _inlineCallbacks(None, gen, Deferred())
    return unwindGenerator
```

storage.py：

```
def DatabaseSafe(method):
    @defer.inlineCallbacks
    @functools.wraps(method)
    def run(self, *args, **kwargs):
        try:
            r = yield defer.maybeDeferred(method, self, *args, **kwargs)
        except cyclone.redis.ConnectionError, e:
            m = "redis.ConnectionError: %s" % e
            log.msg(m)
            raise cyclone.web.HTTPError(503, m)  # Service Unavailable
        except (MySQLdb.InterfaceError, MySQLdb.OperationalError), e:
            m = "mysql.Error: %s" % e
            log.msg(m)
            raise cyclone.web.HTTPError(503, m)  # Service Unavailable
        else:
            defer.returnValue(r)
    return run
```

这三个装饰器都是独立于模块内其他类的单独函数。其中，storage.DatabaseSafe 本身被 defer.inlineCallbacks 装饰。这些都是采用 functools.wraps()制作的装饰器。

笔者在装饰器的学习上花了很多时间。笔者最后发现在 Web 设计中必须使用装饰器来实现权限管理等任务；否则，业务逻辑会过于复杂，而且存在太多冗余代码。

所以，笔者先比较了 Twisted、Cyclone、Tornado 和 Flask 的装饰器，然后参考 Flask 的 RBAC/ACL 实现，再参考 Cyclone 的其他装饰器语法在 Cyclone 中实现了 RBAC 装饰器。

```
def permission_required(permission):
    def decorator(f):
        @functools.wraps(f)
        def decorated_function(self, *args, **kwargs):
            if not self.current_user:
                raise cyclone.web.HTTPError(403)
            return f(self, *args, **kwargs)
        return decorated_function
    return decorator

def admin_required(f):
    return permission_required(permission.ADMIN_PRIVILEGE)(f)
```

普通的 RBAC，比如在最简单的 IoT 设备云中，可以仅仅定义系统管理员和注册用户两级角色。如果系统复杂，则将权限划分为小颗粒，比如将特定方法的权限保存在数据库中。这通过带参数的 permission_required(permission)来实现。不过越复杂的 RBAC/ACL，访问数据库的次数可能就越多。可以适当加载一部分常用的权限定义作为全局变量或保存在缓存中以减少访

问数据库的次数。

3.3　HOWTO：并发运行模型

服务器端和嵌入式系统中的点对点通信存在很大的不同点。服务器端的一个逻辑端口比如80/443 端口要同时连接多个客户端，并且要保持和许多客户端的通信顺畅。换而言之，服务器端必然采用某种并发运行的设计。实现方案可以是单线程异步 I/O，也可以是多线程、多进程和协程。

现在采用得比较多的是单线程异步 I/O 或协程，然后在多核处理或者多台处理器上运行多进程设计。举例来说，Node.js、Twisted 本身都是异步 I/O 设计，运行在单一线程中。每个核心负责运行一个线程，多核主机就是多进程。此外，由于多进程是可以在多个物理/虚拟主机中分布式运行的，所以多核主机的集群也是多进程。

第 2 章中已经介绍过了多进程、多线程，现在我们增加一个协程。

3.3.1　协程

Coroutine，一个比线程更加轻量的并发设计。它在 Lua、Go 等语言中开始普及。在 Go 语言中，Go 语句就是 Goroutine（协程启动）的意思。常见的子程序（routine），也是我们常说的函数，其实是协程的特例。协程可以通过 yield 来调用其他协程。通过 yield 方式转移执行权的协程之间不是调用者与被调用者的关系，而是彼此对称、平等的。常见子程序的实现基于堆栈，采用 FILO，即后进先出方式获得控制权；而协程按照各自需要确定交出控制权给系统。在 Python 中可以利用生成器（generator）来实现协程。

前面介绍过生成器和 yield 关键字。任何包含 yield 关键字的函数都会自动成为生成器（generator）对象，里面的代码一般是一个有限或无限循环结构。每当第一次调用该函数时，会执行到 yield 代码为止并返回本次迭代结果，yield 指令起到的是 return 关键字的作用。然后函数的堆栈会自动冻结在这一行。当函数调用者下一次利用 next()、generator.send()或 for-in 来再次调用该函数时，就会从 yield 代码的下一行开始，继续执行，再返回下一次迭代结果。通过这种方式，迭代器可以实现无限序列和惰性求值。

利用 yield 关键字会自动冻结函数堆栈的特性。两个函数 f1()和 f2()中各自包含了 yield 语句。主线程先启动 f1()，当 f1()执行到 yield 的时候，暂时返回。这时主线程可以将执行权交给 f2()，执行到 f2()的 yield 后，可以再将执行权交给 f1()，从而实现了在同一线程中交错执行 f1()和 f2()。f1()与 f2()就是协同执行的程序，即协程。f1()和 f2()处于对等地位。

generator_coroutine.py：

```
#!/usr/bin/env python

import random

def genData():
    return random.sample(range(10), 4)

def consume():
    moving_sum = 0
    data_seen = 0
    while True:
        print("wait for to consume")
        data = yield
        data_seen += len(data)
        moving_sum += sum(data)
        print 'Consumed, average %f'%(moving_sum/float(data_seen))

def produce(consumer):
    while True:
        data = genData()
        print 'Produced %s'%repr(data)
        consumer.send(data)
        yield

consumer = consume()
consumer.send(None)
producer = produce(consumer)

for i in range(10):
    print ('Producing...')
    next(producer)
```

著名的异步 I/O Web 框架 Tornado 在 V3.0 之后引入@gen.coroutine 装饰器就是采用协程的设计。第三方扩展包 gevent 对 Python 做了修改，提供了比较完善的协程支持。

gevent

gevent 是一种基于协程的 Python 网络库，它使用了 greenlet 提供的、封装了 libev 异步事件循环的高层同步 API。

gevent 提供的编程接口非常类似于传统线程模型，但底层却采用了异步 I/O。而且这一切对用户是透明的。用户可以继续使用这些传统的 Python 模块，比如用 urllib2 去处理 HTTP 请求，它会用 gevent 替换那些普通的阻塞的套接字操作。从 1.0 开始，gevent 底层采用的 C++库从 libevent 切换到了 libev。gevent 特点如下：

- 基于 libev，更快的时间循环。

- 基于 greenlet 轻量级的执行单元。
- socket 支持 SSL。
- 通过 c-ares 或线程池来执行 DNS 查询，c-ares 是异步 DNS 请求的 C 语言库。
- 重用 Python 标准库 API。
- 支持标准的或第三方的同步阻塞 socket 库和模块。

gevent_demo.py：

```
#!/usr/bin/env python
"""Simple server that listens on port 16000 and echos back to the client.
Connect to it with:
  telnet localhost 16000
Terminate the connection by terminating telnet (typically Ctrl-] and then 'quit').
"""
from __future__ import print_function
from gevent.server import StreamServer


# this handler will be run for each incoming connection in a dedicated greenlet
def echo(socket, address):
    print('New connection from %s:%s' % address)
    socket.sendall(b'Welcome to the echo server! Type quit to exit.\r\n')
    # using a makefile because we want to use readline()
    rfileobj = socket.makefile(mode='rb')
    while True:
        line = rfileobj.readline()
        if not line:
            print("client disconnected")
            break
        if line.strip().lower() == b'quit':
            print("client quit")
            break
        socket.sendall(line)
        print("echoed %r" % line)
    rfileobj.close()

if __name__ == '__main__':
    # to make the server use SSL, pass certfile and keyfile to the constructor
    server = StreamServer(('0.0.0.0', 16000), echo)
    # to start the server asynchronously, use its start() method;
    # we use blocking serve_forever() here because we have no other jobs
    print('Starting echo server on port 16000')
    server.serve_forever()
```

3.3.2 I/O 模型

多线程、多进程和协程的目的是实现计算资源的复用。I/O 复用让应用程序同时监控多个 I/O 端口以判断 I/O 操作是否可以进行，实现达到 I/O 的时间复用。这两者有区别，但是在网络编程上有交叉点，因为 I/O 的时间复用也会导致计算资源的复用。所以在网络编程时，它们会被放在一起讲解。

按照《UNIX 网络编程》（*UNIX Network Programming*，Prentice Hall 出版）一书中的划分方法，软件系统的 I/O 模型可以分为五种：

- 阻塞性 I/O；
- 非阻塞性 I/O；
- I/O 复用；
- 信号 I/O；
- 异步 I/O。

如果将 I/O 操作分为数据（等待）就绪和数据复制（处理）两个阶段，那么以上五种 I/O 模型会是以下这样的。

3.3.2.1 阻塞性 I/O

这是最为简单的 I/O 模型。系统会等待某个 I/O，直至所需数据到达；在数据处理过程中等待，直至处理完毕。在数据就绪和数据处理两个阶段，系统会一直等待，并阻塞其他 I/O 操作。在处理完毕后再进行下一个 I/O 请求。

3.3.2.2 非阻塞性 I/O

系统对每次的 I/O 请求，都会立即返回系统。如果没有数据到达，则返回错误。只有 I/O 数据到达后，才会通过某种通知方式进行数据处理。非堵塞性 I/O 与堵塞性 I/O 相比，其数据就绪阶段是非堵塞的。

3.3.2.3 I/O 复用

将多个 I/O 请求传给 I/O 复用系统调用如 select 或 poll，这些 I/O 复用调用会不断地检查多个 I/O 请求的数据就绪情况，一旦有某个 I/O 数据就绪，则进行数据处理。I/O 复用主要是堵塞在 I/O 数据等待阶段。

3.3.2.4 信号 I/O

信号 I/O，即 SIGIO。信号处理函数调用后立即返回系统，数据就绪后，系统会通过 SIGIO 信号通知数据处理进程，即何时可以开始数据复制处理。该模式主要由 BSD UNIX 实现。

3.3.2.5　异步 I/O

异步 I/O 是最快捷的 I/O 模型，线程仅仅注册一次数据操作，由操作系统内核通知何时 I/O 操作完成。该模式主要由 Windows IOCP 实现。

按照 POSIX 的划分法，除了异步 I/O，其余都是同步 I/O，都会在数据处理阶段堵塞进程。虽然 Windows 在服务器领域的份额不如以前，但是 Windows 中的 IOCP 是完全的异步模型，其性能超过其他服务器的实现。在 Linux/UNIX 中，对于文件 I/O 也有 GCC aio、Linux native aio 和 libeio 三种异步 I/O 实现。

3.3.2.6　常见 I/O 复用模型比较与实现

所谓 I/O 复用是指单一线程中“同时”支持多路 I/O 请求。软件系统的 I/O 复用依赖于操作系统和虚拟机/解释器的底层实现。常见的 I/O 复用模型主要有 select、poll、epoll 和 kqueue，其中 kqueue 是 I/O 复用和 SIGIO 的混合体；而 IOCP 是全异步 I/O，其总是作为 I/O 复用模型的比较基准。常见的 I/O 复用模型参见表 3-6。

表 3-6　常见 I/O 复用模型

名　　称	OS/VM	时间复杂度	工作模式
select	Linux	O(*n*)	LT
poll	Linux	O(*n*)	LT
epoll	Linux	O(1)	ET
kqueue	BSD	O(1)	ET
IOCP	Windows	O(1)	ET

此外，还有两个经常使用的设计模式：Reactor 与 Proactor。其中 Reactor 在系统触发回调函数时，通知可以进行 I/O 操作；而 Proactor 则是通知 I/O 操作已经完成。在网络通信中，socket I/O 复用采用何种模型往往会影响框架的性能表现。流行的 C/C++高性能网络软件库如下。

- libevent：名气最大、应用最广泛、历史悠久的跨平台事件驱动通信库。
- libev：与 libevent 相比，其设计更简练，性能更好，但对 Windows 支持不够好。
- libuv：Node.js 的底层异步框架，支持 UNIX/Linux 和 Windows IOCP。

一些网络编程框架如 Twisted/Tornado 分别使用了上一代的异步或 I/O 复用模型，但是现在出现了基于 libuv 的设计，可以直接替代原始设计。所以，现在 Python 的运行速度也变得飞快。

在提升系统计算性能和 I/O 复用方式上有太多的选择。读者需要阅读相关文档，并做一些代码测试，尤其在同样配置情况下，用压力测试来测试出代码的性能和瓶颈，并选择适合自己应用的并发编程模式。

3.4 HOWTO：日期与时间

在任何面对全球部署的应用中，除了本地化之外，时间与日期的表达和转换也是设计中的重要组成部分。在物联网设备中，可以使用 RTC 芯片，但是没有温度补偿的 RTC 依然会发生温漂，依然需要依靠 NTP 服务来校准时间。使用过 NTP 服务的开发者知道，这个协议有时候会遇到一些网络问题导致配置失败。所以，需要为物联网设备定义一个备用的私有授时协议，提高设备授时的成功率。

Python 与时间相关的标准库有 time 和 datetime。第三方库还有 pytz 及其他软件库。Python 的 time/datetime 库中时间和日期的表达形式有四种：

- time struct_time 对象，即由年月日时分秒等构成的元组。
- datetime 对象，datetime 模块使用的内部结构。
- string 对象，表达时间的字符串。
- timestamp，UNIX 时间戳，为浮点数类型。一般 UNIX 时间戳表达从 1970 年 1 月 1 日开始的秒数，是整数类型。但是 Python 还附带毫秒数，所以是浮点类型。与其他系统如 MySQL 对接时需要注意这一点。

```
>>> import time, datetime
>>> time.localtime()
time.struct_time(tm_year=2016, tm_mon=7, tm_mday=18, tm_hour=13, tm_min=24, tm_sec=
23, tm_wday=0, tm_yday=200, tm_isdst=0)
>>> datetime.datetime.now()
datetime.datetime(2016, 7, 18, 13, 23, 52, 484000)
>>> time.ctime()
'Mon Jul 18 13:23:28 2016'
>>> time.time()
1468819471.843
```

以上分别代表根据当地时间产生的不同时间类型：time 元组、datetime、time 字符串、UNIX 时间戳。

3.4.1 类型转换

这四种类型以 time 元组为主，datetime、时间戳和 time 字符串为辅进行转换。许多场合需要使用 UNIX 时间戳和时间字符串表达。比如与网页渲染及 SQL 查询有关的设计中都需要时间字符串。此时需要进行类型转换，转换方法参见表 3-7。

表 3-7　日期时间相关转换方法表

源类型/ 目标类型	time 元组	datetime	UNIX 时间戳	time 字符串
time 元组		datetime(t_tp)	time.mktime(t_tp)	time.strftime(format,t_tp)
datetime	dt.timetuple()		N/A	dt.strftime()
时间戳	time.localtime(ts) time.gmtime(ts)	datetime.fromtimestamp(ts) datetime.utcfromtimestamp(ts)		N/A
字符串	time.strptime(str, format)	datetime.strptime(str, format)	N/A	

可以看到，标注为 N/A 的没有直接转换方式，而是必须要借助 time 元组做一次转换：

```
>>> import time,datetime
>>> ts = time.time()
>>> ts
1468821161.828
>>> time.strftime("%Y%m%d%H%M%S",time.localtime(ts))
'20160718135241'
>>> time.localtime(ts)
time.struct_time(tm_year=2016, tm_mon=7, tm_mday=18, tm_hour=13, tm_min=52, tm_sec=
41, tm_wday=0, tm_yday=200, tm_isdst=0)
>>> y,m,d,h,_m,s,_,_,_ = time.localtime(ts)
>>> y,m,d,h,_m,s
(2016, 7, 18, 13, 52, 41)
```

要获得昨天、今天和明天，可以使用 datetime.date 和 datetime.timedelta 方法。

```
>>> from datetime import date, timedelta
>>> date.today()
datetime.date(2016, 7, 20)
>>> today = datetime.today()
>>> today
datetime.datetime(2016, 7, 20, 20, 3, 12, 513000)
>>> oneday = timedelta(days=1)
>>> yesterday = datetime.today() - oneday
>>> yesterday
datetime.datetime(2016, 7, 19, 20, 3, 12, 513000)
>>> type(today)
<type 'datetime.datetime'>
```

在实际工程的数据采集、压缩归档、定时任务、统计分析等场景中，会反复使用当天、指定日期和时间、一周内、一月内、一年内的数据检索和处理，有必要专门设计一个常用的帮助类来简化这些任务中所需要使用的字符串和时间戳类型转换。

3.4.2 时区的处理

时区不仅仅涉及地理，还涉及行政区管理。我国虽然幅员辽阔，但却是使用单一时区的行政区。美国采用四个标准时区。多时区国家还有俄罗斯、加拿大、巴西、印尼、澳大利亚、哈萨克斯坦、墨西哥和刚果民主共和国。如果加上夏令时、冬令时，以及半小时的时区（如印度、朝鲜）以及尼泊尔的 15 分钟时区，算法实现还是挺复杂的。关于时区与行政关系，可以查看知乎上的相关问答。

time 和 datetime 中已经支持了时区的概念，但是不完整。我们还需要依赖于第三方模块 pytz 或 python-dateutil。请安装使用 PyPI 官网版本。

```
pip install pytz python-dateutil
```

REPL 测试：

```
>>> import time,datetime
>>> dir(time)
['__doc__', '__name__', '__package__', 'accept2dyear', 'altzone', 'asctime', 'clock',
 'ctime', 'daylight', 'gmtime', 'localtime', 'mktime', 'sleep', 'strftime', 'strptim
e', 'struct_time', 'time', 'timezone', 'tzname']
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', '__doc__', '__name__', '__package__', 'date', 'datetime', 'da
tetime_CAPI', 'time', 'timedelta', 'tzinfo']
>>>
>>> imprt pytz
>>> pytz.country_timezones('CN')
[u'Asia/Shanghai', u'Asia/Urumqi']
```

在实际工程中，服务器所在的时区、客户端/设备端所在地的时区与 UTC/GMT（格林尼治时间）是有差异的。比如 GMT 时区是+0，中国境内统一为+8。但是，客户端/设备端却有可能分散在世界各处。设备无法获知自身位置，也无法上报服务器，系统在数据采集、统计分析时就会出现问题。现在许多产品都是全球流通，所以这个问题不得不考虑在内。我们需要根据设备的 IP 地址来判断其所处的时区，必要时需要对设备进行远程授时。

由于服务器、客户端/设备端和 GMT 基准时间是不同的，设计者如何保证使用者的使用习惯，同时保证时间的一致性，这是一个很大的挑战。可以采用 GMT 校准服务器和客户端/设备端的时间，以保证时间的一致性。保障使用者的习惯，即尊重当地时区的计时，则需要服务器在授时信息中包含时区信息。

```
>>> import time,datetime
>>> imprt pytz
>>> dt = datetime.datetime.now()
>>> tz = pytz.timezone(pytz.country_timezones('cn'))
>>> tz
```

```
<DstTzInfo 'Asia/Shanghai' LMT+8:06:00 STD>
>>> dtz = tz.localize(dt)
>>> dtz
datetime.datetime(2016, 7, 19, 13, 38, 35, 469000, tzinfo=<DstTzInfo 'Asia/Shanghai'
 CST+8:00:00 STD>)
>>> pytz.country_timezones('US')
[u'America/New_York', u'America/Detroit', u'America/Kentucky/Louisville', u'America
/Kentucky/Monticello', u'America/Indiana/Indianapolis', u'America/Indiana/Vincennes
', u'America/Indiana/Winamac', u'America/Indiana/Marengo', u'America/Indiana/Peters
burg', u'America/Indiana/Vevay', u'America/Chicago', u'America/Indiana/Tell_City', u
'America/Indiana/Knox', u'America/Menominee', u'America/North_Dakota/Center', u'Ame
rica/North_Dakota/New_Salem', u'America/North_Dakota/Beulah', u'America/Denver', u'
America/Boise', u'America/Phoenix', u'America/Los_Angeles', u'America/Anchorage', u'
America/Juneau', u'America/Sitka', u'America/Metlakatla', u'America/Yakutat', u'Amer
ica/Nome', u'America/Adak', u'Pacific/Honolulu']
>>> pytz.timezone('America/Los_Angeles')
<DstTzInfo 'America/Los_Angeles' LMT-1 day, 16:07:00 STD>
>>> la = pytz.timezone('America/Los_Angeles')
>>> la
<DstTzInfo 'America/Los_Angeles' LMT-1 day, 16:07:00 STD>
>>> datetime.datetime.now(la)
datetime.datetime(2016, 7, 18, 22, 52, 36, 619000, tzinfo=<DstTzInfo 'America/Los_An
geles' PDT-1 day, 17:00:00 DST>)
>>> latm = datetime.datetime.now(la)
>>> latm
datetime.datetime(2016, 7, 18, 22, 56, 36, 547000, tzinfo=<DstTzInfo 'America/Los_An
geles' PDT-1 day, 17:00:00 DST>)
>>> la.normalize(latm)
datetime.datetime(2016, 7, 18, 22, 56, 36, 547000, tzinfo=<DstTzInfo 'America/Los_An
geles' PDT-1 day, 17:00:00 DST>)
```

由于历史原因，pytz 里的 tzinfo 和 Python tzinfo 不一致。这里有个古怪的时间偏移量。实际上不仅是 Asia/Shanghai 时间有偏差，其他地点也有偏移量。所以需要使用 localize 方法来产生正确的时区数值。Python pytz 中有一个 normalize 方法，专门用于解决跨越时区产生的误差以及夏令时处理。

实际上，要处理好时区问题，还有更多问题需要解决。读者可自行阅读 pytz 的文档。

3.5　Python 版本迁移

Python 3 的发展很快。在不久的将来，或许主流的 Python 程序都会逐渐迁徙到 Python 3。

3.5.1 Python 2 与 Python 3 的区别

对于 Python 2 和 Python 3，不同的开发者对于其兼容性评价完全不同，这完全取决于他们使用的各类库的兼容性。但是需要记住：Python 2 和 Python 3 本质上是两种语言，但是我们可以将代码设计为 Python 2/3 兼容。它们之间的区别如下：

- print 不再是语句，而是内置函数，即 print'abc'必须写成 print('abc')；
- 没有旧式类，只有新式类；
- 整数除法为浮点数除法，即 5/2=2.5，而非 2；
- 采用了新的字符串格式化方法 format 取代%；
- xrange 改为 range，并返回迭代器对象；
- 不等于采用!=，而非<>；
- 整数统一为 int；
- 异常语法有改变；
- exec 从语句变成函数；
- 类库变化；
- import 采用绝对路径导入；
- 通过 input()解析用户输入；
- 加入 async 协程；
- 用 bytes 替代 str，原生支持 unicode，删除单独的 unicode 类。

3.5.2 Python 2 到 Python 3 的流程

先将用户代码升级到 Python 2.5 以后的版本，最好是 Python 2.7；最好进行覆盖测试。

（1）了解 Python 2/Python 3 之间的区别；

（2）使用 Modernize 或者 Furturize 来更新代码；

（3）使用 Pylint 来确认升级到 Python 3；

（4）使用 caniusepython 3 来寻找 Python 3 兼容的瓶颈；

（5）使用 tox 测试 Python 2/Python 3 兼容性。

以上各个第三方包请使用 pip 安装。

此外，由于 Python 3 并非以性能优化为第一考虑要素，因此 Python 3 的运行速度可能会比 Python 2 要慢。

3.5.3 多个 Python 版本共存

在许多时候，我们不得不在多个 Python 版本间切换。而在不同操作系统和工程中，实现方

式有些不同。但总的来说，需要实现的目的如下：

- 多个 Python 虚拟机，包括 V2/V3、32/64 位、CPython/PyPy/Jython 等；
- 多个 Python 虚拟机对应的软件库和 pip；
- Python 文件的关联（Windows）。

3.5.4　virtualenv

在 virtualenv 推出之前，开发者使用两个不同版本 Python 的时候需要对系统路径和环境变量进行配置和切换。而现在，我们可以通过 virtualenv 隔离不同版本的 Python 环境。

```
pip install virtualenv
virtualenv --no-site-packages venv
```

一般来说，不同 virtualenv 是为了单一应用而配置的干净的 Python 环境，之后可以采用 pip 在 virutalenv 环境中安装第三方库。virtualenv 创建的虚拟环境与主机的 Python 环境完全无关，主机配置的库不能在 virtualenv 中直接使用。你需要在虚拟环境中利用 pip install 再次安装配置后才能使用。

virtualenvwrapper

virtualenv 是为了创建虚拟环境，而 virtualenvwrapper 是为了管理虚拟环境。

3.5.5　Windows 多个版本共存

在 UNIX/Linux 中，Python 的用户脚本中可以利用 shebang 声明来显式地指定运行的 Python 虚拟机。Windows 则无法实现这一点，因为 Windows 的文件类型关联是全局有效的。我们当然可以使用手动切换和底层重命名来实现。但是我们还可以使用 batch、doskey 和 regedit 来实现自动切换。

3.5.5.1　手工配置

分别安装 Python 2.7.11 和 Python 3.4 之后，默认路径是：

```
C:\Python27\python.exe
C:\Python34\python.exe
```

pip 的默认路径是：

```
C:\Python27\Script\pip.exe
C:\Python34\Script\pip.exe
```

一些教程建议将这些 python.exe 改名以区分版本，但改名不算是很好的方式。在 cmd 终端下采用 doskey 做命令替换比较合理。

```
doskey python27=C:\Python27\python.exe $*
doskey python34=C:\Python34\python.exe $*
```

其中$*代表参数，否则会出现 python.exe 无法携带参数的现象。

在 cmd 终端中可以采用 python27 或 python34 直接调用对应版本的 Python。至于 pip，笔者发现升级 pip 之后，对应版本会出现 pip.exe、pip2.exe、pip27.exe 文件，内容一致。升级后的 pip 已经为我们配置好了版本。

升级 pip 的方式是：

```
python -m pip install --upgrade pip
```

切换 Python 环境一般发生在进入 cmd 后，可以制作一些专门的 batch 文件实现。但是 Python 文件与 Windows 的关联需要另外的脚本去配置。因为文件关联是全局的，所以仍需要来回切换。

3.5.5.2 pywin

pywin 是 pyenv 在 Windows 中的对应软件。它利用了 MSYS/MINGW32 的 shell 以及 DOS/PowerShell batch 实现了多个版本 Python 的切换。其内置 pyassoc，而且还支持 Windows Python 文件关联的切换。

Python 版本是单个会话有效，即打开单一命令行窗口（Bash shell 或 PowerShell）后切换版本则有效，关闭后失效。pyassoc 是 Windows 操作系统全局有效。切换后，单击 py 文件会一直使用对应版本的 Python 虚拟机去运行，直至被修改。

注意 pywin 不是 pywin32。前者是切换 Windows 中 Python 环境的工具，后者是 Python 调用 Win32 系统 DLL 的软件包。

3.5.6 Linux 多个版本共存

在 Linux 发行版如 Ubuntu 中默认安装了 Python 2 和 Python 3，切换也挺麻烦。pyenv 能够帮助我们来管理不同版本的关系。

pyenv 有以下功能：

- 进行全局的 Python 版本切换；
- 为单个项目提供对应的 Python 版本（通过 virtualenv 插件）；
- 使用环境变量能重写 Python 版本；
- 能在同一时间在不同版本间进行命令搜索。

其拥有以下特点：

- 只依赖 Python 本身；
- 将目录添加进$PATH 即可使用；
- 能够进行 virtualenv 管理。

3.6　其他常见技巧

本节介绍一些难以归类但却可能会用到的 Python 技巧。

3.6.1　常数类型的模拟

C 和 Java 语言中都有常数类型：

```
const float PI = 3.1415926;    // In C/C++

public static final String CONSTANT_NAME = "It rocks";    // In Java
```

Python 的作者和维护者可能认为任何不提供改动方法的对象都可以被视为常数。此例中的 PI 就是通常意义上的“常数”。

可是，Python 作为动态类型语言，总有方法可以修改常数 PI 的定义。这在某些应用中存在风险。而且 Python 除了本身的内置类型外，没有常数类型。所以，必须使用一个专门的类来模拟。其实现方式多种多样。这实际需要满足两个要求：作用域和不可修改。

```
class _const:
    class ConstError(TypeError): pass
    def __setattr__(self, name, value):
        if self.__dict__.has_key(name):
            raise self.ConstError, "Can't rebind const (%s)" %name
        self.__dic__[name]=value

import sys
sys.module[__name__]=_const()
```

在上面的例子中，常数类的__setattr__方法中会直接抛出异常，禁止修改。

3.6.2　枚举类型的模拟

与常数类型类似，C 语言里有一个枚举类型；Python 2 必须使用其他类型来模拟。其中枚举类型往往用于定义状态常数。而 Python 等动态语言是没有常数概念的，所以需要采用类来模拟。其实现方式同样非常多。在 Python 3.4 之后，增加了 Enum 枚举类。

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3

print(Color.red) # Color.red
```

3.6.3 开发自定义模块

随着代码越写越多，会逐渐积累一些私有模块。这些代码不一定是通用功能性代码，而有可能是面向特定工程的模块。比如在笔者经手的物联网设计中，使用 Twisted/Cyclone/Celery/Plan 以及其他 Python 组件构建整体服务，但是与工程有密切关联的配置和私有传输格式都可以采用自定义模块的方式来统一。

- 配置信息模块：easyiot.py。
- 数据持久模块：easydb.py。

如果在 Twisted/Cyclone/Celery/Plan 文件夹中分别放置配置信息模块和数据持久模块，在应用代码中 import（导入）这两个模块即可使用。如果要将这些模块做成包（package），则需要构建一个单独的文件夹，并放置一个空白文件：__init__.py。

```
easyiot/
    easyiot.py
    easydb.py
    __init__.py
```

编写 setup.py：

```
from distutils.core import setup

setup(
    name = 'easyiot',
    version = '0.1',
    py_modules = ['easyiot'],
    author = 'Allan K Liu',
    author_email = 'allankliu@163.com',
    description = 'EPIC EasyIOT'
)
```

运行 setup.py：

```
allankliu@ubuntu-server-vm:~/demo/easyiot$ sudo python setup.py bdist
running bdist
running bdist_dumb
running build
running build_py
creating build/lib.linux-i686-2.7
copying easyiot.py -> build/lib.linux-i686-2.7
installing to build/bdist.linux-i686/dumb
running install
running install_lib
creating build/bdist.linux-i686/dumb
creating build/bdist.linux-i686/dumb/usr
creating build/bdist.linux-i686/dumb/usr/local
```

```
creating build/bdist.linux-i686/dumb/usr/local/lib
creating build/bdist.linux-i686/dumb/usr/local/lib/python2.7
creating build/bdist.linux-i686/dumb/usr/local/lib/python2.7/dist-packages
copying build/lib.linux-i686-2.7/easyiot.py -> build/bdist.linux-i686/dumb/usr/loca
l/lib/python2.7/dist-packages
byte-compiling build/bdist.linux-i686/dumb/usr/local/lib/python2.7/dist-packages/ea
syiot.py to easyiot.pyc
running install_egg_info
Writing build/bdist.linux-i686/dumb/usr/local/lib/python2.7/dist-packages/easyiot-0.
1.egg-info
Creating tar archive
removing 'build/bdist.linux-i686/dumb' (and everything under it)
```

安装自定义模块：

```
allankliu@ubuntu-server-vm:~/demo/easyiot$ sudo python setup.py install
running install
running build
running build_py
running install_lib
copying build/lib.linux-i686-2.7/easyiot.py -> /usr/local/lib/python2.7/dist-packages
byte-compiling /usr/local/lib/python2.7/dist-packages/easyiot.py to easyiot.pyc
running install_egg_info
Removing /usr/local/lib/python2.7/dist-packages/easyiot-0.1.egg-info
Writing /usr/local/lib/python2.7/dist-packages/easyiot-0.1.egg-info
```

检测是否可以正常导入自定义模块：

```
allankliu@ubuntu-server-vm:~/demo/easyiot$ python
Python 2.7.3 (default, Dec 18 2014, 19:03:52)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import easyiot
>>> dir(easyiot)
['DmConf', '__builtins__', '__doc__', '__file__', '__name__', '__package__', 'os',
'socket', 'sys']
>>> exit()
```

3.7　Python 与其他语言

本章介绍 Python 的各种实现方式以及与各种语言的接口。所谓实现方式（implementation），指的是使用何种编程语言来构建 Python 解释器的方式。而接口指的是 Python 程序与这些编程语言之间的调用方法。

虽然 C/C++/Java 等拥有大量的衍生版本，但是这些版本往往指的是针对特定硬件或者操作

系统平台的版本。还没有哪种语言像 Python 一样被多种高级语言实现过其解释器。

由于 Python 可以被其他语言实现，所以这些 Python 实现往往和底层语言有着紧密的联系。通过阅读本章，读者可以充分理解 Python 为何被称为胶水语言。

越来越多的拥有其他编程语言背景的程序员可以选择自己熟悉语言的相对应 Python 实现，并与原有代码进行对接。

Python 的不同实现

Python 是一门跨平台的脚本语言，Python 规定了语法规则用于实现 Python 语法的解释器，再实现标准库就可以构成完整的 Python 运行环境了。Python 语法解释器可以有多种语言实现，从而使得 Python 可以与不同语言进行沟通，实现不同的目的。这些实现虽然都可以运行 Python 代码，但是它们彼此之间还是有差异的。

1．CPython

CPython，即 Classic Python。后来为了与其他实现区别开来，才称之为 CPython，即 C 语言实现的 Python，这也是最常见的版本。CPython 会将源文件（.py 文件）转换成字节码文件（.pyc 文件），然后由 CPython 虚拟机运行。CPython 往往使用 ctypes 扩展来调用 C 语言编写的外部库。

2．Jython

其原名为 JPython，Java 语言实现的 Python。根据其官方描述，Jython 是 Java 语言进入 21 世纪后最强大的武器。

Jython 的功能如下：

- 动态编译为 Java 字节码，可以产生最高性能而无须牺牲交互性；
- 可以在 Jython 中扩展 Java 类，允许有效地使用抽象类；
- 可选的静态编译，可以支持 applet、servlet、bean 和其他 Java 类；
- bean 属性，可更加容易地利用 Java 包；
- Python 语言，拥有清晰语法的强大语言，与 Java 一样支持 OOP。

Jython 适合的场景如下：

- 原型；
- 检查 Java 类，这是 Jython 设计的核心目的之一；
- 访问 bean 属性；
- 充分利用现有的 Java 库，整合使用；
- 出色的嵌入式脚本语言。

Jython 与 CPython 的差异如下：

- Jython 大部分使用 Java 编写，也使用了 Java Native Runtime，但有安全限制；

- Java 7/8；
- 编译成$py.class 文件；
- 通过 JFFI 接口，可使用 Java 或者 C 扩展，未来使用 CFFI、ctypes 和 C 扩展 API；
- 真正的多线程（这一点和 CPython 的 GIL 完全不同）；
- Java 垃圾回收，包括标准 Java GC 和 Python 2.7 的通用 API。

Jython 主要用于以下场景：

- 操作系统不提供 CPython，只有 Java 可以用。比如 Sun 工作站，或者是某些嵌入式设备。
- 不得不用一些 Java 的包来提供功能。所以，只能用 Jython 来调用 JAR 包。
- 采用 Python 的测试框架对 Java 类进行快速测试。

Jython 是一种完整的语言，是 Python 语言在 Java 中的完整实现，而不仅仅是 Java 翻译器或 Python 编译器。Jython 也有很多从 CPython 中继承的模块库。总体来讲，Jython 继承自 Python 的支持包不如 CPython 多。不过，纯 Python 包都可以在 Jython 上安装使用。

相比于 CPython，Jython 与 Java 语言之间的互操作性要远远高于 CPython 和 C 语言之间的互操作性。最有意义的是，Jython 不像 CPython 或其他任何高级语言实现，它提供了对其底层语言 Java 的一切访问。Jython 不仅提供了 Python 的库，同时也访问所有的 Java 类。这使得 Jython 拥有一个（或者我们可以称之为两个）超大的软件资源库。

在 Python 代码中可以直接使用 Java 代码库，这可以使用 Python 方便地为 Java 程序写测试代码；更进一步，可以在 Python 中使用 Swing 等 Java 图形库编写 GUI 程序。不仅如此，Jython 开发者可以直接访问一些目前仅提供 Java 接口的企业级软件库和应用。比如 Hadoop 采用 Java 编写，开发者可以提供 CPython 可用的 Streaming 接口，但却不如 Java 完整。所以 Hadoop 也提供 Jython 的 JAR 包，可以直接访问 Hadoop Java 类。

Java 的类库已经相当庞大，再加上 Python 社群提供的库，这对有选择障碍症的人来说也是一件麻烦事。不过 Jython 最初是为了 Java 程序员熟悉 Python 而设计的，所以这些群体会很自然地选择 Java 类库。虽然笔者学习 Java 的时间比 Python 更早，但却不算是合格的 Java 程序员。所以，笔者在开发工程时主要于 Python 库中寻找。

Jython 会将 Python 代码动态编译成 Java 字节码，然后在 JVM 上运行转换后的程序。这意味着此时 Python 程序与 Java 程序没有区别，只是源代码不一样。此外，Jython 基于 JVM，所以它的性能或许比 CPython 要高；避免 GIL 也可以使得它在多线程应用上要比 CPython 优秀。理论上 Jython 的并发特性会更好，因为 Jython 可以利用 Java JIT 优化运行速度，并可以充分利用多核处理器资源。

每种 Python 实现都会自带 benchmark 例程，可以用来测试 Python 解释器性能。但是笔者测试下来，运行速度最快的是 PyPy，其次是 Cython，再次是 CPython，而 Jython 仅为 CPython 的 1/8。读者可以自行编写测试程序，看看在多核或者其他配置下的 Jython 性能表现。

3．PyPy

PyPy 是使用 Python 实现的 Python。确切地说，是由 rPython 实现的，而 rPython 是 Python 的子集。

看上去这好像有些逻辑问题：用 Python 实现的 Python，那么最初的 Python 是如何实现的？这种自我螺旋式发展在计算机历史上出现过好多次了。如果读者有兴趣、有时间体验一次，建议构建一次交叉编译器，这样就可以充分体会这种自我螺旋式发展的意义和困难了。

PyPy 是 Armin Rigo 开发的 Python 语言的动态编译器。作为 Psyco 的后继项目，PyPy 的目的是实现动态编译。PyPy 开始只是研究性质的项目，但是目前成为主流的 Python 加速方案之一。

PyPy 与 CPython 相比更加灵活，且其易于使用和实验，还可以指定特定功能在不同情况中的实现方法，很容易实施。该项目的目标是：让 PyPy 比 CPython 更为容易地适应各个项目，方便功能裁剪。

最新的 PyPy 版本是 PyPy 3 2.1 beta1，兼容 Python 3.2.3。该版本可以运行在 x86 平台的 Linux 的 32 位与 64 位以及 Mac OS X 和 Windows 32 位平台中，而基于 ARM 平台的版本正在开发中。它支持 Python 语言的所有核心部分以及大多数 Python 语言标准库函数模块。至于 PyPy 与 CPython 的区别，读者可以去看其官网的兼容性页面。

PyPy 还提供了 JIT 编译器和沙盒功能，因此其运行速度比 CPython 要快，并可以安全地运行一些不被信任的代码。PyPy 还有一个单独的支持微线程的版本。PyPy 声称可以突破 Python 中非常著名的 GIL（Gloabl Interpreter Lock）全局解释器锁定瓶颈，实现并行计算。

奇怪吧，Python 比 C 要慢，PyPy 却比 CPython 要快！

究其原因是 PyPy 使用了速度奇快的 JIT 编译器。所谓 JIT（Just In Time），是将字节码编译成本地机器码（Native Code）的编译器。但是其在生产环境中使用还有一些限制因素，如对于 SSL 的支持和某些模块的兼容性，需要综合考虑。但是请继续关注 PyPy 的开发进展，因为 PyPy 可能是未来 Python 的趋势。

在实际生产环境中，笔者考虑在服务器端某些环节中采用 PyPy 替代 CPython，并配合 libuv/libev 异步 I/O 以实现最大的性能提升和连接并发数。

4．Python for .NET

Python for .NET 是一个开源项目。这是 CPython 实现的.NET 托管版本，与.NET 库和代码有很好的互通性。

Python for .NET 是让 Python 程序员可以与.NET 公共语言运行库（CLR，Common Language Runtime）进行无缝整合的软件包，它为.NET 开发者提供了强大的应用脚本支持。使用该软件包可以对.NET 应用进行脚本化，也可以使用 Python 构建.NET 应用，重用 C++/C#/VB/JScript 编写的.NET 服务和组件。

该软件包并不将 Python 作为 CLR 语言来实现，即它并不从 Python 源码中产生中间托管码（IL）。它是将 CPython 解释器和.NET 运行环境进行了整合。这种方式，既可以使用 CLR 服务，也可以继续使用现有的 Python 代码和 C 扩展，同时又保证了 Python 代码的执行速度。

5．IronPython

IronPython 是一种在.NET 和 Mono 上实现的 Python 语言，由 Jim Hugunin 所创造。他是 Python NumPy 的创建者，也是 Jython 和 IronPython 的创建者，此外他还是 Java AspectJ（AOP，面向切面编程）扩展的联合设计师。他曾经于 2004—2010 年在微软工作，其主要工作是 IronPython。

不同于 Python for .NET，IronPython 是 Python 的 C#实现。与 Jython 类似，它将 Python 代码编译成 C#中间代码（IL），然后运行，旦与.NET 语言的互操作性也非常好。IronPython 的应用场景更多的是为 C#应用增加脚本能力。

除了主流 Java/.NET 平台，还存在着许多其他流派的 Python 实现。

6．Stackless Python

Stackless Python 是 Python 的一个增强版本。支持微线程编程模式，并避免传统 Python 线程的性能和复杂度问题。Stackless Python 的微线程扩展是低开销的轻量级设计。其最主要的目的是提供并发编程扩展。据测，Stackless Python 的并发特性惊人。

CPython 的一个局限就是每个 Python 函数调用都会产生一个 C 函数调用。这意味着同时产生的函数调用是有限制的，因此 Python 难以实现用户级的线程库和复杂递归应用。一旦超越这个限制，程序就会崩溃。

Stackless 的 Python（包括 PyPy）实现突破了这个限制，一个 C 栈帧可以拥有任意数量的 Python 栈帧。这样你就能够拥有几乎无穷的函数调用，并能支持巨大数量的线程。Stackless 的唯一问题就是它要对现有的 CPython 解释器做重大修改。所以，它几乎是一个独立的分支。另一个名为 Greenlets 的项目也支持微线程。它是一个标准的 C 扩展，因此不需要对标准 Python 解释器做任何修改。

7．Cython

请读者注意拼写，是 Cython，不是 CPython。

Cython 是一种优化的静态编译器，用于编译 Python 语言和扩展后的 Cython 编程语言。它使得编写 C 扩展如同编写 Python 一样容易，而且运行速度飞快！

Cython 使得用户具有可以同时利用 Python 和 C 两种语言的能力：

- Python 代码可以与 C/C++代码互相调用；
- 通过静态类型声明，Python 代码的性能很容易调整到 C 语言级别；
- 使用混合型源码调试可以调试 Python、Cython 和 C 代码；
- 具有与大型数据集的有效沟通，如可以使用多维 NumPy Array；

- 生态完整，构建应用较容易；
- 可以有效地整合老式代码，设计底层或者高性能代码。

跨平台 NUI 组件 Kivy，就是基于 Cython 的设计。

8．Numba

Python 著名的科学计算包 NumPy 的创始人 Travis Oliphant 推出了 Numba 项目。它能够将处理 NumPy 数组的 Python 函数通过 LLVM JIT 编译为原生机器码运行，从而提升了上百倍的运行速度。

9．IPython

IPython 是一个交互式 Python shell，但比 Python 官方版本 IDLE 增加了一些特性：

- 自动补全；
- 自动缩进；
- Bash 指令；
- 代码自动保存（ipython notebook）。

IPython 不仅仅是一个 Python shell，它还有：

- 采用 Jupyter 内核；
- 解释器可以嵌入到用户项目中去；
- 支持交互数据可视化和 GUI 工具箱；
- 并行计算的高性能工具箱。

IPython 本质上是 CPython 的另外一种增强型封装，而且在 IPython/Jupyter 分离后的新设计中，不再局限于 Python，还支持 Haskell、R、Julia、Lua、Scala 语言编程。

10．RubyPython

Ruby 和 Python 有许多类似的地方。RubyPython 是 Ruby/Python 翻译器之间的桥梁。它使用 FFI 在 Ruby 应用进程中嵌入一个 Python 翻译器，并提供了封装、转换和调用 Python 对象与方法的手段。使用 FFI 可以规整 Ruby 和 Python 虚拟机之间的数据，并调用 Python。

笔者感觉其开发目的是为了让 Ruby 程序充分利用 Python 的第三方库。

11．PyObjC

PyObjC 为 Python 和 Objective-C 两种语言之间提供了接口。Python 可以通过 PyObjC 来使用已有的 Objective-C 代码，反之亦然。PyObjC 最主要的用处是在 Mac OS X 系统下使用纯 Python 语言来开发 Cocca GUI 应用程序。

12．Pyjs（Pyjamas）

Pyjs 是用于 Web 和桌面的 RIA 开发平台，是 Python 到 JavaScript 的编译器、AJAX 框架和 Widget API。谷歌 GWT 是一种参考 Java AWT 构建的 Java 软件包，可以转为 JavaScript 代码。Pyjs 相当于谷歌 GWT 的 Python 版本，同时支持桌面应用开发：Pyjs Desktop。

13．Brython

Brython 的设计目的是将 Python 作为替代 JavaScript 的前端网页脚本。以下是一个例子：

```
<html>
<head>
<script src="/brython.js"></script>
</head>
<body onLoad="brython()">
<script type="text/python">
def echo():
    alert(doc["zone"].value)
</script>
<input id="zone"><button onclick="echo()">clic !</button>
</body>
</html>
```

使用 Brython 必须加载 brython.js，让这个脚本来翻译 Python 语法。它本质上是一个 Python 的 JavaScript 实现。这个设计很有趣，用 JavaScript 实现的 Python，并尝试替代 JavaScript 的位置。至少我们可以用 Python 来构建网页交互动作和动画了。

3.8　Python 语言扩展

作为一门功能强大的高级脚本语言，Python 不仅有着丰富的功能，而且具备良好的可扩展性，并被屡屡成功地应用于各类大型软件系统的开发过程中。

Python 扩展（extensions）其实就是一个软件库。一般来说，能够被整合或者导入到其他 Python 脚本的代码，都可被称为扩展。扩展可以使用纯 Python、Java，也可以使用 C/C++编译型语言来编写。

3.8.1　C 语言扩展 Python

在物联网开发中，我们将大量使用 pyserial 包。pyserial 包中的 Java/Jython 部分就采用了 JavaComm 扩展。此外，需要高速计算的加密模块和二进制处理模块中采用了 C 扩展，需要性能加速的库如 lxml 也是 C 扩展。

Python 扩展的原因

使用 Python 扩展的原因有以下几种：

- 添加额外的非 Python 功能，如新的数据类型，或者将 Python 嵌入其他语言。
- 性能瓶颈，效率提升。解释型语言一般比编译型语言慢，两者有机结合是一种很好的

方式。Python 程序员可以使用 C 或 C++来进行功能性扩展。这样既可以利用 Python 方便灵活的语法和功能，又可以获得与 C/C++几乎相同的执行性能。

- 源码私密性也是扩展 Python 的理由之一。采用 C/C++和 Python 混合编程除了性能原因，还可以满足知识产权保护、对核心代码进行加密的要求。

在本章特地介绍 C 扩展，是因为在之后的物联网设备端设计，我们必须依靠 C 扩展来解决许多特定的问题，甚至要自己改造现有的扩展库。C 和 Python 结合的解决方案不止一种：

- Python C API，最初的解决方案。通过引入 Python.h 头文件，对应的 C 可以直接使用 Python 数据结构，将 C 语言编译成链接库文件（Linux 中的 so 文件，Windows 中的 DLL 文件）用于 Python 直接调用。
- ctypes，Python 标准库，用于封装 C 程序，可以直接写 Python 代码加载 so/ DLL 文件，无须源文件。想要使用现有 C 类库，ctypes 是最直接的方式。
- Cython，既可以调用 C 文件函数，也可以调用 Python 函数；可兼容 NumPy 大量包含 C 扩展的库。使用场景一般是针对算法和过程进行优化。
- cffi，类似于 ctypes，是 ctypes 在 PyPy 中的实现，同时兼容 CPython。可以直接在 Python 代码中编写 C 代码。
- swig，wrapper 的一种，可将 C++语言代码封装转换为 Python C API，并生成 so 文件。其支持 Java/Ruby/Lua 等。
- sip，另一种 wrapper，由 swig 发展而来，接口文件略有差异。
- Boost::python，随 boost 库一起发布，可全自动封装项目。
- Py++，调用 boost.python 完成项目绑定。
- PyCxx，轻量级的封装库。
- Weave，性能高。
- Pyrex：性能较高，接近 Python 语法。

前四种解决方案算是比较常见的方式。其中，Python C API 最基础，使用方法最烦琐。其余的方法都是在其基础之上的各种简化和分支方法，各有其应用场景。在嵌入式应用中，C API 扩展、cffi 和 ctypes 往往都会使用到。ctypes 使用起来也比较烦琐，但在访问一些闭源 Windows DLL 库时尤为重要。

以下实例用于说明 C 语言扩展 Python 的流程，适用于桌面版 Linux（Ubuntu） + GCC + Python。

hello.c：

```
#include <Python.h>

static PyObject* helloworld(PyObject* self)
{
```

```
    return Py_BuildValue("s", "Hello, world. Returned as string.");
}

static char helloworld_docs[] =
    "helloworld( ): Comments for helloworld\n";

static PyMethodDef helloworld_funcs[] = {
    {"helloworld", (PyCFunction)helloworld,
     METH_NOARGS, helloworld_docs},
    {NULL}
};

void inithelloworld(void)
{
    Py_InitModule3("helloworld", helloworld_funcs,
                   "Extension module example!");
}
```

在 Python 文档中，有 Py_InitModule/Py_InitModule2/Py_InitModule3/Py_InitModule4 共四个函数，此外还有 64 位版本等，请参考其具体用法。

setup.py：

```
#!/usr/bin/env python
from distutils.core import setup, Extension

setup(name='helloworld', version='1.0',  \
      ext_modules=[Extension('helloworld', ['hello.c'])])
```

运行 setup.py 后，该脚本将 helloworld 扩展放置到了 Python 的安装路径中去了。这样其他 Python 脚本就可以引用这个包。

extensions-demo.py：

```
#!/usr/bin/python
import helloworld

print helloworld.helloworld()
```

3.8.2　ctypes 访问 Windows DLL

本节不是使用 C/C++来扩展 Python，而是使用 Python 访问现有动态链接库。ctypes 是 Python 访问操作系统中动态链接库的方式，包括 Linux 的 so 和 Windows 的 DLL。Linux 中虽然是开源的，但是出于性能上的考虑，许多 Python 封装了各种 lib*.so，构成了新的 Python 扩展库。相比之下，Windows 上的闭源商业系统比较多。许多设备供应商的二次开发接口往往仅提供 DLL 和头文件，甚至连文档都没有，更不要提某些 DLL 库需要进行逆向工程的情景了。所以 Python

ctypes 的最大价值是访问 Windows DLL 库和闭源商业软件库，例如访问某些硬件的 DLL 库，这是一种必须掌握的技能。

首先创建一个 Windows DLL 工程，并使用 Visual Studio 编译。MingW 和一些其他的 C++ 编译器也可以编译出 Windows DLL 库，不过需要反编译工具来检查生成的 DLL 兼容性。具体情况请查看本章延伸阅读内容。

Python 调用 Windows DLL 导出的接口。

```
#!/usr/bin/env python
# test.py
import ctypes
dll = ctypes.windll.LoadLibrary('test.dll')
test = ctypes.c_init(dll.fntest())
print(test_result.value)
```

3.8.3 Jython 访问 Java 类

CPython 无法直接访问 Java，必须利用 Jython 访问 Java 类。这也是 Jython 开发的目的。在 JVM 中运行 Python 代码可以带来许多额外的收益。首先到 Jython 官网下载安装包：jython_installer-2.7.0. jar，并采用以下指令安装：

```
java -jar jython_install-2.7.0.jar

java -jar jython_install-2.7.0.jar --console
```

后者适用于命令行安装。安装后会建立一个新文件夹，C:\jython2.7.0。

```
C:\jython2.7.0>java -jar jython.jar
Jython 2.7.0 (default:9987c746f838, Apr 29 2015, 02:25:11)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on java1.8.0_66
Type "help", "copyright", "credits" or "license" for more information.
>>> copyright
Copyright (c) 2000-2015 Jython Developers.
All rights reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 2000 The Apache Software Foundation.
All rights reserved.

Copyright (c) 1995-2000 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

```
>>>exit()
```

REPL 测试：

```
Jython 2.7.0 (default:9987c746f838, Apr 29 2015, 02:25:11)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on java1.8.0_66
Type "help", "copyright", "credits" or "license" for more information.
>>> from java.util import Date
>>> d = Date()
>>> print d
Wed Mar 16 13:29:45 CST 2016
>>> from java.util import Random
>>> print dir(Random)
['__class__', '__copy__', '__deepcopy__', '__delattr__', '__doc__', '__ensure_finali
zer__', '__eq__', '__format__', '__g
etattribute__', '__hash__', '__init__', '__ne__', '__new__', '__reduce__', '__reduce
_ex__', '__repr__', '__setattr__', '
__str__', '__subclasshook__', '__unicode__', 'class', 'doubles', 'equals', 'getClass
', 'hashCode', 'ints', 'longs', 'nex
tBoolean', 'nextBytes', 'nextDouble', 'nextFloat', 'nextGaussian', 'nextInt', 'nextL
ong', 'notify', 'notifyAll', 'seed',
 'setSeed', 'toString', 'wait']
```

在安装后的 C:\jython2.7.0\Demo 中有一些 Jython 的实例，可以使用以下语法运行测试：

```
cd \jython2.7.0
java -jar jython.jar Demo\awt\Colors.py
java -jar jython.jar Demo\swing\ObjectTree.py
```

著名的大数据 Hadoop 采用 Java 编写。所以，除了兼容所有语言的 Streaming API 之外，Jython 是 Python 程序员访问 Hadoop 的最直接途径。

移动应用如 Android 中，也主要采用 Java 编程。而在 Kivy/Pyjnius 工程中，可以使用 Python 或者 Cython 访问 Android 的 Java 类，真是条条大路通罗马啊！

3.8.4　IronPython 访问.NET

IronPython 的安装程序包含了二进制文件、Python 标准程序库、用于 Silverlight 的 IronPython 和一个教程。此外，Pyc.py 工具用于将 IronPython 应用程序编译成二进制文件。Ipy.exe 可以用来执行 IronPython 程序，对于熟悉 Python 语言及探索.NET 程序来说非常有用。

IronPython 允许在 C#代码中加载 Python 类、调用 Python 类函数。

TestIronPython.cs：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
```

```
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using IronPython.Hosting;

namespace TestIronPython
{
    public partial class Form1:Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            PythonEngine scriptEngine = new PythonEngine();
            scriptEngine.AddToPath(Application.StartupPath);
            scriptEngine.Execute(textBox1.Text);
        }
    }
}
```

Execute 是按照字符串进行逐句解释运行的。如果要调取整个 Python 文件，则需要使用 ExecuteFile(@"filename.py")方法。Python 只可调用 C/C++编译的 DLL，无法直接调用 C#编译的 DLL，需要通过 IronPython 来调用。

IronPython_CallDll.cs：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace IronPython_CallDll
{
    public class Dll_static_methods
    {
        public static int Add(int x, int y)
        {
            return x + y;
        }
    }

    public class Dll_normal_methods
    {
        private int sum = 11;
        public int SumUp(int m)
```

```
        {
            sum = sum + m;
        }

        public void ShowMsgBox()
        {
            global::System.Windows.Forms.MessageBox.Show(sum.ToString());
        }

    }
}
```

ironpython_demo.py：

```
import clr
clr.AddReferenceByPartialName("System.Windows.Forms")
clr.AddReferenceByPartialName("System.Drawing")
from System.Windows.Forms import *
from System.Drawing import *

clr.AddReferenceToFile("IronPython_CallDll.dll")
from IronPython_CallDll import *

a = 100
b = 20
s = Dll_static_methods.Add(a,b)
MessageBox.Show(s.ToString())

di = Dll_normal_methods()
di.SumUp(200)
di.SumUp(30)
di.ShowMsgBox()
```

类库中的静态方法可以直接调用，而普通方法需要先定义实例，再访问实例方法。即便在同一个类中，以上原则依然有效。如果能够充分利用 IronPython 的能力，可以实现节省研发资源的目的。笔者就在一些用户项目中重用了原有的 C#工程代码。

3.9　Python 加速

Python 简单易用，开发速度快。但是 CPython 执行速度不算快，数量级低于 C/Go，甚至还低于 Java/JavaScript。其一直被人诟病是开发快、运行慢。如何使用不同方法加速 Python 运行速度是许多 Python 分支的开发目的之一。Python 的加速工具有许多，总的开发趋势是利用编译器或者 JIT 编译器进行加速，或者利用计算硬件实现并行计算：

- Cython，利用 C 编译器加速；
- Numba，利用 LLVM JIT 加速；
- PyPy，利用 LLVM JIT 加速；
- LLVMpy，利用 LLVM JIT 加速；
- Pyston，利用 JIT 加速；
- PyCUDA，利用 CUDA 实现并行计算；
- PyOpenCL，利用 APU/GPU/FPGA 硬件加速。

关于 Python 的性能加速，本章延伸阅读部分和网络上有一篇名为 *Python Performance Tips* 的文章，列举了 Python 性能优化的许多实现方法，值得一读。

3.9.1 PyPy

前面已经介绍过了，PyPy 是常见的 Python 加速方法。据某些测评报告称，采用 PyPy 加速后的 Twisted 是 Node.js 性能的两倍。在本章延伸阅读部分中罗列了不同程序员对比 PyPy 和 Node.js 的性能表现。

根据笔者自己的测试，PyPy/CPython/Jython 的性能基数分别是 8：1：0.5，测试基准程序是 Python 各个实现自带的 pystone.py 脚本。

Python 核心开发者 Yury Selivanov 有篇技术博文，对比了多种网络编程方法，包括：

- asyncio-streams；
- Tornado；
- curio-stream；
- Twisted；
- curio-socket；
- uvloop-streams；
- gevent；
- asyncio
- Node.js；
- uvloop；
- Golang 协程。

在同一测试基准和功能要求下，基于 libuv 的 uvloop 速度和 Golang 差不多。此外，Node.js 的表现超越原始的 asyncio/gevent/Twisted 和 Tornado。根据 Yury 提供的数据推测，如果采用 PyPy 加速，Twisted/Tornado 的确有可能比 Node.js 要快。这印证了之前 PyPy/Node.js 的测评报告。

3.9.2　Cython

Cython 是 Python 的 C 语言扩展，准确地说 Cython 是专门用来写 Python 扩展库的语言。它采用 C 编译器编译代码，速度的确飞快。

3.9.3　PyCUDA

CUDA（Compute Unified Device Architecture）是由 NVIDIA 推出的 GPU 通用计算平台，是高性能并行计算以实现计算加速的典型手段。而对应的 PyCUDA 让用户可以使用简单易用的 Python 来使用 CUDA 平台。由于 CUDA 实际上是在 GPU 中运行的，应用中最大的瓶颈为数据在主存储器和显示存储器间的移动与复制。PyCUDA 在和 HPC 相关的计算中，可以让用户专注于算法，而非 GPU 细节。PyCUDA 可以为某些计算密集型任务提供快速普及的手段。可以实现 3D、VR、AR、噪声定位和消除噪声等应用，也可以为 DSP 算法验证提供一个合适的平台。只是目前嵌入式平台中只有 NVIDIA 一款处理器 Tegra-K1 可选。手机等消费品平台不支持 CUDA。

3.9.4　PyOpenCL

OpenCL 类似于 CUDA，但是其适配平台更加广泛，适用于 CPU/GPU/DSP/FPGA 等。许多芯片都已经提供了 OpenCL 支持，但很可惜 Android 领军企业谷歌不支持 OpenCL，所以，其在移动端也遇到了瓶颈。OpenCL 对应的 Python 封装为 PyOpenCL。

3.9.5　Theano

Theano 是用于定义、优化和计算数学表达式的 Python 库，并且可以利用 GPU 加速计算：

- Theano 支持定义、优化、计算多维数组在内的数学表达式；
- 与 NumPy 紧密结合；
- GPU 加速，使用 32 位浮点可以加速数据密集型运算，某是 CPU 计算的 140 倍；
- 高效符号差分计算；
- 针对速度和稳定性进行过优化；
- 动态产生 C 代码；
- 单元测试和自验证。

自 2007 年以来，Theano 开始在大规模科学计算中使用，并可用于机器学习/深度学习课程教学。

3.9.6　Nuitka

Nuitka 是一个 Python 的替代编译器，可以无缝替代和扩展 Python 的解释和编译工作。它

可以执行编译代码，并能用兼容方式将目标代码一起编译。开发者可以自由地使用所有 Python 模块库和第三方扩展，Nuitka 将其编译成 C 级别的程序。这些优化工作是为了避免 Python 执行产生的不必要开销。

使用编译器和 JIT 加速后，Python 程序的性能得到很大提升，终于可以摆脱运行速度慢的帽子了。配合合适的 C++异步库，如 libuv 作为底层实现，其整体的系统运行性能令人惊喜。

3.10 本章小结

本章在第 2 章 Python 的基础知识之上，介绍了常见的物联网设计任务的实现，并简单地讲解了一些常见的知识难点。其中，多种语言实现和性能加速会在日后物联网开发中扮演非常重要的角色。在使用这些技术的同时，读者可以充分理解 Python 在系统集成领域起到的胶水作用。

第 4 章

嵌入式系统开发

本章将针对传统嵌入式系统的整体设计做一个全局性的系统介绍，内容涵盖了硬件、固件（及软件）和生产相关内容。

- 硬件包括：SPICE 仿真、数字逻辑设计和可编程器件、微控制器、微处理器、传感器、执行器和通信 IC 等。
- 软件包括：编程语言演进、C/C++驱动编程、编译器、操作系统和 RTOS 内核、中间件、通信堆栈。
- 生产和设计杂项：安全、固件更新、工具和参考设计。

由于物联网特别注重连接性，所以第 5 章将介绍嵌入式系统连接能力的多样性和对应编程接口，以及相关 Python 编程实例，并演示如何利用这些接口进行系统对接以完成设备端系统开发的过程。

虽然在物联网系统中，归属于设备端的嵌入式系统开发只是其中一个环节，但是需要积累大量的应用和电子工程领域的知识。嵌入式系统开发本身的环节就很长，包括以下环节。

- 系统设计：根据数据传速率、功耗、地理分布情况选择合适的 WSN 网络协议和 IC、路由器（边缘服务器）系统架构和 IC、其他系统接口和 IC，生产模式和维修模式固件设计，以及生产工位设计与接口。
- 模具设计：因为要避免空间干涉，所以必要时可采用机械 MCAD 与电子 ECAD 联合设计，避免空间干涉。
- 硬件设计：根据系统设计，落实到 IC 的原理图和 PCB 布局，要考虑散热和电磁兼容性，并预留调试、批量生产、维修所需接口和测试点。
- 固件设计：根据硬件、成本、许可证来选择 C 编译器、中间件和调试手段，并自制测试脚本。
- 软件设计：可选项目，根据主机或操作系统定制 UI 和功能集。

即使具备多年嵌入式系统开发经验的工程师，也经常会遇到各种各样的技术陷阱。这主要是因为从底层到高层有各种技术依赖性。技术加上成本、商业条件约束，有时候会造成令人进

退维谷的状态。如果考虑芯片供货、价格、架构、样品、小批量，批量生产以及产品良率的因素，产品开发的风险非常高。技术选型伊始，许多开发者从一开始就朝着错误的方向进行，比如选择了错误的接入方式等，接下来会步步错。

在嵌入式系统中，软硬件的边界选择也是一个问题。一些系统逻辑如按键去抖动、ADC 采样移动平均、串行协议、PWM 输出，甚至 USB 通信，既可以采用硬件，也可以采用软件实现。软件实现的代价是软件复杂度和功耗，硬件实现的代价是依赖性、供货和成本。如何兼顾两者则是系统设计师的经验和功力所在。

在物联网的各个环节中，越靠近硬件，迭代成本越高、代价越高；越靠近终端软件，迭代成本越小，迭代速度越快。为了适应这种趋势，各个行业出现了各种解决方案，尽量降低成本，加快迭代速度。典型的例子就是键盘虚拟化，硬件生态平台化，操作系统标准化。在新品开发时，开发周期和开发人力成本的重要性远远超越产品 BOM 成本。

本章将尝试对于物联网提供一些比较切合市场现实的解决方案。让读者少踩一些“坑”。

物联网与传统设备不同，其对于通信安全和与通信相关的附加功能如固件更新等方面提出了新要求。本章最后会给出一些实际的建议和方案。

4.1 嵌入式系统硬件分类

笔者将市面上的半导体芯片分成若干大类：

- 主控芯片，用于控制周边外围芯片，运行程序，实现应用目的。
- 传感器，用于采集物理量数据。
- 执行器，用于执行程序命令对外部施加作用。
- 通信芯片，用于芯片间组网联网的各种芯片。
- 可编程逻辑，用于实现芯片间耦合或者以上芯片的原型开发。
- 辅助类芯片，可以细分为存储器、电源、模拟电路等。

笔者先介绍几种市场上可以看到的主控制器结构。自从半导体行业将独立的 ALU 等逻辑功能模块整合在一起后，出现了各种各样的架构设计：MCU、MPU、DSP……百花齐放。

4.1.1 MCU

图 4-1 中的 DIP40 封装集成电路是最经典的微控制器：Intel P8051AH。采用 NMOS 工艺，单片 8 位微控制器，带 32 组 I/O，2 组定时器，5 组中断，2 级中断优先级，4KB ROM（MASK 掩膜工艺），128 字节 RAM。虽然当年 Intel 退出了 MCU 市场，主要关注 CPU，但却授权其他厂家继续生产 8051 架构 MCU。8051 架构 MCU 已成为 MCU 的工业标准之一。

图 4-1　Intel P8051AH

8051 之所以成为经典，一方面是因为 8051 在 8048 的架构基础上做了许多改进，最大寻址空间可以达到 64KB；另外一方面是因为其可以外扩 EPROM/SRAM，成为一个小系统板。最后一点，Intel 面向教育界提供了大量的资源。由于 8051 价格低廉，与 8086/80286 相比其更加适合作为教学使用，因此其成为许多大学计算机原理的硬件实验平台。

与 8051 同时代的 MCU 多是 4/8/16 位处理器架构，通常采用哈佛结构，将 RAM/ROM、GPIO、ADC、PWM、I2C、SPI 等外设整合到片内。即便外扩 RAM/ROM，也不支持 MMU。所以，这些 MCU 上运行的往往是 RTOS，无法运行 Linux 的复杂操作系统。当然，一些外扩的 MCU 如 ARM7TDMI 可以运行经过剪裁后的 Linux 变种（如 uCLinux）。

随着 ARM/MIPS 等将一些低端内核如 Cortex-M0/3/4 导入 MCU 后，MCU/MPU 的分界日渐模糊。

4.1.2　MPU

MPU 的 CPU 单元相对 MCU 而言性能更强，支持外扩 RAM，支持 MMU，程序加载在 RAM 中运行，往往使用 Linux/CE/Android 等操作系统，（但也需要区分采用最小系统还是完整系统的 Linux）。因为 Python 在这些系统需要依赖底层的服务来实现，所以底层服务的变化会影响到高层 Python 的实现和包的依赖。

开发 MCU/MPU 往往需要额外的开发工具：ICE/JTAG 仿真器，以及 Debugger 调试器和烧录器。开发者往往会在不同的工具依赖性上耗费大量精力和资金：比如开发某些 MCU，需要特定的 JTAG/SWD/ICE、定制的 RTOS，以及定制的 C 编译器、库函数和中间件。商业版本的全功能编译器和中间件价值不菲。

4.1.3　DSP

数字信号处理器是一类特殊的处理器。DSP 比较适合不间断的流式数据处理。它的寻址方式、指令集和存储结构等与传统 MCU/MPU 存在较大差异。虽然许多 CPU、MCU 中增加了 DSP 指令集，但是许多计算密集型任务依然需要 DSP 芯片或者内核的参与。主要 DSP 供应商是得

州仪器（IT）和 ADI 公司。DSP 芯片和 IP 供应商数量较少，架构也彼此不一致。其开发门槛较高。同样，DSP 算法往往也都需要开发者掌握深厚的数学知识。

目前，DSP 产品线受到了来自 FPGA 阵营的硬件 DSP 的挑战。因为在相当多的场景下，FPGA 数据处理速度更快。

4.1.4 SMP

SMP（对称多处理器）是最简单的多核心处理架构，是通过复制多个同样的处理器，共享内存子系统和总线结构，实现紧密耦合的多处理器系统。现在大多数主流桌面、服务器和移动计算处理器都完成了从单核到多核的演进。手机移动处理器动辄八核心 ARM Cortex 处理器，计算能力已经超过市场所需。

SMP 可以很好地实现多进程计算，不过需要操作系统和编程语言以及应用程序能够充分利用多核架构。

4.1.5 异构大小核

所谓异构大小核是一种将高性能和低功耗 CPU/MCU 进行混搭的架构设计。异构大小核的目的是，利用高性能处理器来负责复杂和计算密集型任务，利用低功耗处理器来降低功耗、处理低速 I/O 任务和实时任务，或负责专门的 DSP 算法。主要搭配方式如下。

- CPU+MCU：一些机顶盒方案中采用 MIPS/ARM+8051 混搭，MIPS/ARM 运行 Linux/Android，8051 负责遥控解码。
- CPU+CPU：ARM 的 big.LITTLE 大小核设计是比较典型的。Samsung/MTK 的 SoC 中采用 Cortex-A53/A7/Cortex-A15 搭配。
- CPU+DSP：TI DaVinci/OMAP 产品线是典型的 DSP+ARM/Cortex 架构。DSP 用于基带和音频，早期功能手机上大量采用此架构。
- MCU+MCU：NXP 的 LPC4357 支持 Cortex-M4 + Cortex-M0 双核处理器。

智能手机中的 A53/A15 都是为了让手机在不同功耗模式中进行切换，以实现最大限度节省电池寿命。现在其已经成为许多平台的主流选择。MCU+MCU 的配置，往往针对专门应用而设计。

市场上的四旋翼多采用开源飞行控制器，图 4-2 是开源市场比较热门的 pixhawk 无人机飞控模块，其由著名的 3DR 公司推出。pixhawk 第一代采用了 STM32F427 + STM32F103 高低搭配两枚 MCU 来做飞控。STM32F427 为 Cortex-M4F 内核，内置单精度浮点协处理器，适用于飞控所需要的姿态计算；而 STM32F103 是作为协处理器以做后备控制器使用的，以防止 F427 失控。pixhawk 第二代采用了 Intel Edison 平台，该设计由 3DR 公司主推。

图 4-2　pixhawk 开源四旋翼飞行控制器

在 pixhawk 之前的 APM 飞控模块采用的是 Atmel 的 ATMEGA2560+ATMEGA328 做高低搭配，大小核设计思路与 M4F+M3 组合一脉相承。无人机、车联网应用场景非常适合 LPC4357 或者 LPC54100 使用，均为 Cortex-M4F/Cortex-M0+搭配。而且其价格仅仅是 F427 的四分之一。在许多基于传感器和浮点计算新型应用中，可以大量使用此类混合型 MCU。这些应用包括可穿戴设备、独立 VR 眼镜、机器人等。

4.1.6　FPGA 原型

FPGA 和 CPLD 都归类于可编程逻辑器件。FPGA 的架构和使用场景相对更广，并形成了寡头型的市场局面：Xilinx、Altera、Lattice、Actel、Microsemi、Avago 和 Cypress 是现存的供应商，其中，前两家占据了主要市场份额。

MPU/MCU 和 FPGA 有彼此整合的趋势。FPGA 供应商都有集成软核、硬核、固核 IP 的 FPGA 产品线；传统 MPU/MCU 供应商也有这些产品，如 Intel Xeon E5 2600 V4 推出内含 FPGA 的产品，而 Atmel 内置 FPGA/CPLD 的产品存在已久。Intel 收购 Altera 之后，会推出新的融合性产品用于 AI。

4.1.7　SoPC

SoPC，即 System on Programmable Chip（片上可编程系统芯片）。用可编程逻辑技术把整个系统设计在一块硅片上，称作 SoPC。SoPC 是一种特殊的嵌入式系统：首先它是片上系统（SoC），即由单个芯片完成整个系统的主要逻辑功能；其次，它是可编程系统，具有灵活的设计方式，可裁减、可扩充、可升级，并具备软硬件在线系统可编程（ISP）的功能。

从 IP 核的提供方式上，通常将其分为软核、固核和硬核这三类。从完成 IP 核所花费的成

本来讲，硬核代价最大；从使用灵活性来讲，软核的可复用使用性最高。与软核实现方式相比，硬核可以把功耗降低 5~10 倍，节约将近 90%的逻辑资源。

4.1.7.1 软核（Soft IP Core）

软核在 EDA 设计领域指的是综合之前的寄存器传输级（RTL）模型；具体在 FPGA 设计中指的是对电路的硬件语言描述，包括逻辑描述、网表和帮助文档等。软核只经过功能仿真，它需要经过综合以及布局布线才能使用。其优点是灵活性高、可移植性强，允许用户自配置；缺点是对模块的预测性较低，在后续设计中存在发生错误的可能性，有一定的设计风险。软核是 IP 核应用最广泛的形式。

比较常见的 Nios II、MicroBlaze/PicoBlaze 就是 Altera/Xilinx 的软核处理器。

4.1.7.2 固核（Firm IP Core）

固核在 EDA 设计领域指的是带有平面规划信息的网表；具体在 FPGA 设计中可以看作带有布局规划的软核，通常以 RTL 代码和对应具体工艺网表的混合形式提供。将 RTL 描述结合具体标准单元库进行综合优化设计，形成门级网表，再通过布局布线工具即可使用。和软核相比，固核的设计灵活性稍差，但在可靠性上有较大提高。 目前，固核也是 IP 核的主流形式之一。

4.1.7.3 硬核（Hard IP Core）

硬核在 EDA 设计领域指经过验证的设计版图；具体在 FPGA 设计中指布局和工艺固定、经过前端和后端验证的设计，并且设计人员不能对其进行修改。不能修改的原因有两个：首先是系统设计对各个模块的时序要求很严格，不允许打乱已有的物理版图；其次是保护知识产权的要求，不允许设计人员对其有任何改动。IP 硬核的不许修改特点使其复用有一定的困难，因此只能用于某些特定应用，使用范围较窄。

硬核 IP 主要有 MIPS、ARM 的 Cortex-M，这些都是比较主流的 IP 核。FPGA 的主要客户群还是做 IC 原型设计而使用，一旦量产后，可以使用 SoC 来替代。除了灵活性和通用性，FPGA 在功耗、成本、封装上都不占优势。

SoPC 推荐 Xilinx 的 Zynq（AP SoC，All Programmable System on Chip）。虽然 Zynq 是硬核 IP+FPGA 的配置，却提供了基于硬件的 DSP 模块，可以视为 MPU+DSP+ FPGA。这已经对 TI/ADI 等构成了实际的威胁。在最新的 VR 直播中，可以选用 Zynq 作为一种平台。图 4-3 是 Digilent 基于 Xilinx Zynq-7020 推出的 PYNQ-Z1。在国内电商平台也有销售。PYNQ-Z1 的最大特点是使用 Python 搭建嵌入式应用。通过将底层硬件进行封装，用户可以直接使用 Python 进行编程，甚至基于浏览器 Jupyter Notebook 就可以编辑工程代码。除了 PYNQ 外，用户还可以使用传统的 Xilinx 工具，如 Vivado、HLS 等。

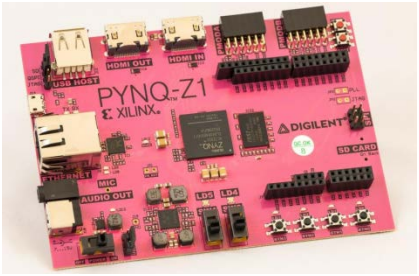


图 4-3　PYNQ-Z1 开发板

4.1.8　GPU

利用 GPU 的多核计算能力，可以实现任务的并发任务处理。和 DSP 类似，GPU 适合计算密集型任务，而且计算通道比 DSP 多。它适合密码验算、2D/3D VR/AR 及图形计算。

由于 GPU 很早就出现了双寡头的局面，GPGPU（General Purpose GPU，通用计算 GPU）标准 CUDA/OpenCL 也主要是 NVIDIA 和 AMD/ATI 在推广。基于现有的架构，CPU 运算需要复杂的序列代码，而 GPU 则可以将其分解为许多简单的计算，实现并行运行。不过，GPU 高性能计算也有自己的问题：

- GPU 计算必须在 GPRAM 和主存储器之间复制数据，这成为系统瓶颈。
- GPU 浮点计算精度不够，仅为单精度浮点。
- GPU 计算适合并行计算，不适合串行化的计算任务。

NVIDIA 已经推出了 HPC 产品线（GPU 公司推出的没有 GPU 功能的并行计算产品线）。而且其和合作伙伴推出了 HPC 集群云服务，GPU 计算将在 AR/VR/AI 开辟出一个全新的应用计算领域。

针对嵌入式应用，NVIDIA 推出了 GPU SoC 和开发套件。图 4-4 是 NVIDIA TX1，为 Maxwell GPGPU 结构，提供 256 个 CUDA 单元、64 位 ARM 处理器，且板载 miniPCIe、eMMC、DDR、SATA，并支持 Linux。

图 4-4　NVIDIA JETSON TX1 开发板

除了上述几种控制器架构外，其余的架构相对少见：如 XMOS 的多核 MCU、Candense/Tensilica 的 Xtensa DPU 处理器、Synopsys DesignWare 视频处理器。随着人工智能应用的增多，出现了更多其他架构。2016 年，中星微就针对视频监控领域推出了 NPU（神经网络处理器）。可以预见，更多颠覆性的架构会不断出现。

4.1.9 哈佛结构和冯・诺依曼结构

在 MCU/MPU 架构设计领域，哈佛和冯・诺依曼结构非常有名。

哈佛结构将数据和代码分开存储，所以数据和指令总线分离。这种分离的结构使得处理器在同一机器周期中同时获得指令和操作数，彼此总线不会发生竞争的状态。带来的附加效应是该结构的数据和指令宽度可以是不同的。最典型的案例莫过于 PIC 单片机，其数据宽度为 8 位，但是指令宽度却有多种宽度，如 12/14/16 位。著名的 8051/AVR 等都是哈佛结构。但这种架构的代码空间和数据空间是分开寻址的，在某些需要更新代码的应用中需要特殊的地址映射逻辑。ARM 在推出 ARM7 的替代产品 Cortex-M 系列 MCU 时也采用该结构的改进型。

冯・诺依曼结构，又称普林斯顿结构。该结构使用同一存储器，将数据和代码放在该存储器的不同位置，经过同一总线传输，所以访问指令和操作数是先后进行的。冯・诺依曼结构推出的年代，程序一度都是单一目的固化代码，让程序可以自我修改以加快迭代速度很重要。冯・诺依曼结构的提出是为了实现可以动态地修改程序。由于使用同一存储器，所以该结构处理器的数据和指令宽度是相等的。Intel 8086/MIPS/ARM7 均采用这种架构。

随着技术的发展，出现了更多改进型和混合型的架构：改进型/增强型哈佛结构和超级哈佛结构等。在改进型哈佛结构中，比如指令和数据虽然在两块存储器（两组总线）中，但是却编码在同一寻址空间中，Cortex-M 就是如此。而现代处理器往往内部有缓存，分为数据缓存和指令缓存，采用哈佛结构；主存储器接口采用的是冯・诺依曼结构。

底层硬件开发者必须熟悉这些计算机架构，通过各种总线来解决系统瓶颈。而且这些结构的优化对于汇编和 C 语言更有意义。许多建立在虚拟机之上的高级语言如 Java、JavaScript、Python，甚至 Lua 这种嵌入式脚本语言，代码必须在数据存储器中得到解释和执行。

4.2 电路原型设计

本节主要叙述 Python 在电路原型设计，包括电路描述仿真、硬件描述和可编程逻辑器件方面的应用。实际上本节是笔者在编写本书收集资料时发现的“彩蛋”章节。MyHDL 让笔者发现，原来 Python 在 HDL/EDA 领域也有了一席之地，而且还可以进行波形仿真和自动测试！笔者继而展开检索，发现 Python 不仅在数字电路方面，而且在 SPICE 方面也有许多应用。如果说

MyHDL 是学术界的研究结果，那么基于 Zynq FPGA 推出的 PYNQ-Z1 开发板，则是厂家对于 Python 在系统原型开发和人工智能领域的殷殷期望了。FPGA 产品线是一个比较特殊的器件，其使用者遍布整个物联网生态链条，从芯片原型开发、小规模定制的嵌入式系统开发，到数据分析、机器视觉、机器学习和人工智能，都有大量应用案例。接下来，我们先了解一下传统芯片设计的流程。

现在许多集成电路都采用塑料封装，所以读者对于 IC 内部缺乏直观感觉。图 4-5 是 Philips Semiconductors 于 1988 年出品的 S87C752 MCU。当时只有 UV-EPROM/EPROM 工艺，这枚 MCU 采用陶瓷封装，开发者可以利用紫外线将内部 EPROM 的代码擦除。

透过石英玻璃窗口可以观察到集成电路内部的基板、IC die 和框架。IC die 切割后往往是方形的。集成电路生产线中有专门的绑定（wire-bonding）机器将 IC die 与框架通过点焊方式连接起来。S87C752 基于 8051 内核，内置 ADC，所以是一枚典型的数模混合型 MCU。如果仔细观察 IC die，可以看到大面积的分布比较规整的 EPROM，以及分布比较杂散的 ADC。读者查阅数据手册，可以了解一下这些外设的实际布局。

图 4-5 中的陶瓷封装采用 2.54mm/100mil 脚距，所以可以大致推算出 IC die 的面积。集成电路单位面积越小，其成本越低。将多枚 IC die 绑定在一个框架封装中可以实现 SiP（System in Package）封装。一些物联网芯片在实现 SoC 芯片之前，可以通过 SiP 来实现系统整合。

图 4-5　Philips S87C752 MCU

嵌入式开发的基础是集成电路。作为基础硬件，集成电路的开发却呈现出软件化的特征。这种变化的原因有许多：

- 硬件描述代码比原理图更加容易实现版本控制、对比、修改。
- 硬件描述代码比原理图更加容易实现大规模集成。

- 软核比硬核更加容易适配到最终版图中，修改更加容易。
- 工程师团队中的软件工程师更多。
- EDA 工具迎合了以上的变化。

笔者曾经遇到过一个小型团队，几位软件工程师设计了一款 IC 卡芯片，基本上就是用了 8051 内核，加上合适的外围 IP，在 IC 代工厂直接流片，封装、测试，然后交付给最终用户。这充分说明了硬件设计软件化的趋势。

4.2.1 集成电路设计流程

IC 设计分为前端设计和后端设计。前端设计是原理和逻辑设计，负责系统级和行为级描述，将寄存器级/门级交由后端实现；后端设计负责将前端设计具体实现，包括版图实现、工艺优化、面积优化，掩膜等……

如果将电路粗略分为模拟电路和数字电路进行划分，行业内会采用不同的设计手段来构建原型、验证、设计和实施。其他的无线、微波、光电、MEMS 都有各自领域的工具，就不介绍了。

集成电路开发流程一般是这样的：首先采用硬件描述语言编写其数字逻辑，进行功能性仿真、逻辑综合、FPGA 适配，以及时序与逻辑门级仿真，最后下载到 FPGA 芯片上进行原型验证。另外一种方式是采用原理图绘制替代硬件描述语言，重复整个流程，下载到 FPGA 芯片中验证，最后交付 RTL（Register Tranfer Level，寄存器转换级）设计。这基本上就是前端设计流程了。

FPGA 验证后，需要和代工厂配合，根据工厂工艺（SPICE 模型）进行版图拼接适配，后端布局工程师的目的是解决各种工艺和寄生参数问题，以及将单个晶圆最小化，并进行后仿真，以及一系列的反复迭代。这基本上就是后端设计流程了。

完成后端设计后，可以安排流片、生产；晶圆（Wafer）生产后，要进行测试、切割、封装、激光标记……切割后的单个裸片被称为 die。在生产过程中由于各种原因（如灰尘）所导致的坏片，被称为 NGD。所以说，集成电路的迭代成本非常昂贵。有些芯片之所以卖得很便宜，其实很可能是由于其节省了部分测试成本，回避了芯片良率的问题。

总的来说，在数字电路中，使用硬件描述语言构建数字电路模型，并在 FPGA 中实现电路原型完成前端设计，然后交由后端设计去做芯片布局。模拟集成电路往往无法切割清楚前后端设计，其大部分需要依赖于电路仿真。其不仅需要在设计电路时进行仿真，而且布局后还需要后仿真。

4.2.2 模拟电路原型设计

笔者的模拟电路功底不算很好，所以在涉及模拟电路设计时，常常依赖 SPICE 仿真工具做

原理验证，并使用面包板做电路实测。

术语翻译

中国国内普遍使用的某些技术术语有时候会造成大家理解上的混乱。比如，“仿真”一词就被用滥了，有两个场景中都在使用“仿真”一词。

- 在模拟（Analog）电路中，使用 SPICE 来仿真（simulate）电路的运行。
- 在调试“单片机软件”时，使用在线仿真器来仿真（emulate）程序运行。

相比较而言，我国港台地区的这一术语比较合理：

- 在类比（Analog）电路中，使用 SPICE 来模拟（simulate）电路的运行。
- 在调试“微控制器固件”时，使用在线仿真器来仿真（emulate）程序运行。

很明显，**模拟**适用于在微机中以纯软件方式运行电路分析程序；而**仿真**适用于以某种硬件替代微控制器在电路（实际运行环境）中运行。而且，**类比**电路也不会与动词**模拟**冲突。但出于对国内学术环境的尊重，笔者也不得不混用这类术语了。

4.2.2.1　SPICE

SPICE（Simulation Program with Integrated Circuit Emphasis）是最为普遍的通用电路级模拟程序，历史悠久。SPICE 的网表格式变成了通常模拟电路和晶体管级电路描述的标准。许多模拟电路元件供应商除了提供器件数据手册，也提供相对应的各种模型：IBIS 模型，S 参数。SPICE 模型一般不向应用级客户提供，而向 IC 设计客户提供。此外，在 SI 信号完整性分析中，SPICE 模型也是不可缺少的。

各软件厂家提供了 Vspice、Hspice、Pspice 等不同版本的 SPICE 软件，其仿真核心大同小异，都是采用了由美国加州大学 Berkeley（伯克利）分校开发的 SPICE 模拟算法。虽说如此，但是各个软件的分析收敛算法和速度有所区别，甚至有的软件公司为这些算法注册了专利。

4.2.2.2　eispice

eispice 参考的是 Berkeley SPICE3 仿真引擎的源码。最初该软件是为了 PCB 级信号完整性仿真而设计的，用于仿真 IBIS 模型、传输线和被动终端，但后来慢慢地扩展到通用电路仿真功能上。图 4-6 是 eispice 的软件截图。

1．仿真引擎

仿真引擎实际上采用 C 编写，并利用 SuperLU 矩阵库来计算 MNA 矩阵。在大多数 IBIS 信号完整仿真上，应该会比 Berkeley SPICE 引擎要快。但因为 SPICE 并不支持 IBIS，所以无法直接比较。

2．SPICE 模型

eispice 包含了标准 SPICE3 的器件模型子集。其目标是兼容 SPICE3F5 版本的所有基础模型。eispice 还原生支持 IBIS 模型、Python 的行为级模型、非线性电容等。

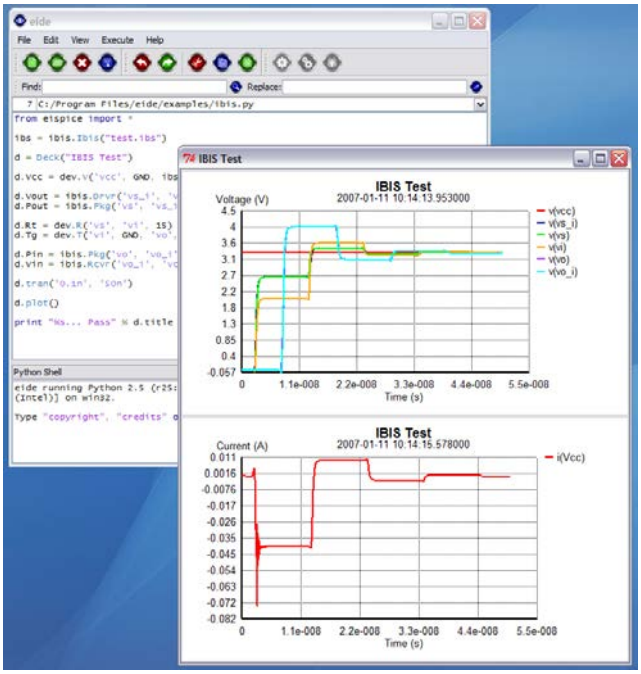


图 4-6　eispice 软件截图

3．Python 前端

除了原生支持 IBIS 模型的开源仿真器，eispice 的独特性还包括其 Python 前端。仿真器本身由 C 语言编写，但 eispice 仿真器被封装成 Python 模块，用户可以使用 Python 来控制仿真和处理结果。对于仅仅熟悉 Berkeley SPICE 而不熟悉 Python 的开发者，可以使用 Python shell 替代 Nutmeg，并利用 Python 脚本作为 SPICE 批处理。

Python 还可以集成多个不同的 Python 模块。配合 4.2.3.3 节中提到的 MyHDL，Python 可以整合 MyHDL 和 eispsice 实现混合模型仿真。

4．绘图引擎

eispice 内置的绘图工具非常简单，不过读者可以使用 Python matplotlib 来补强。

4.2.2.3　PyOPUS

PyOPUS 由流行的免费电路仿真器 SPICE OPUS 的开发团队，即位于斯洛文尼亚首都的卢布尔雅那（Ljubljana）大学电子工程系计算机辅助设计实验室开发。图 4-7 是该软件包配合 matplotlib 的仿真截图。

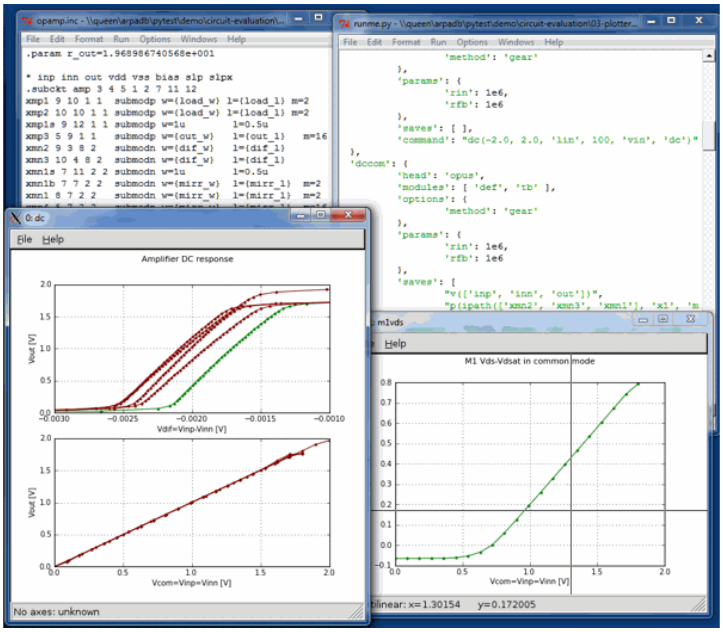


图 4-7　PyOPUS SPICE 仿真演示截图

SPICE OPUS（OPtimization Utilities for SPICE）是一个面向优化循环的通用电路仿真器。该软件在 Berkeley SPICE 源码上针对 Windows/Linux 进行了重新编译，然后加入了乔治亚州技术研究所的 XSpice 混合仿真器。XSpice 代码模型得到了增强，可以从 dll/so 库文件加载代码模型（.cm 文件）。该仿真器还包括一个解释型的编程语言 Nutmeg，用于交互式 SPICE 仿真会话。其开发团队解决了若干内存泄漏的问题，重写了程序的图形部分，并保留了原始的 plot/iplot 命令，且支持其他 SPICE 的脚本。该开发团队还在不断增加新的半导体模型。

4.2.2.4　PySpice

PyPI 官网上的 PySpice 是 Python 3 的扩展包。PySpice 是 Fabrice Salvaire 的个人项目，利用 Python 产生并操作 Berkeley SPICE 电路，进行模拟和仿真输出。在其最新版本中，增加了使用实例，以及使用电路图工具产生电路，然后导入 Python 模型中。Fabrice 最初用它解决以下问题：

- SPICE 语言用于描述电路，但不是一种电路操纵的语言。与此相反，Python 作为一种强大的面向对象动态语言，可以用来操纵和重用电路设计。代价是 Python 的通用编程语法描述电路不如 SPICE。
- Ngspice 在 Berkeley SPICE 基础上增加了数据分析，但它的交互语法和 Tcl（Tool Command Language）模块已经有些过时。不过，Python 的科学计算框架如 NumPy/matplotlib 足可以与 MATLAB 相比较。

- Ngspice 源码来自 Berkeley SPICE，该源码基础有些老旧，而且文档缺失，维护困难。PySpice 可以作为一个实验性的扩展。

功能如下：

- 仅支持 Ngspice。
- 采用 SPICE 类似的面向对象方式描述电路。
- 自带目录和索引的元件库和模型管理器。
- 实验性的 SPICE 网表解析器，可以使用 KiCAD 作为原理图工具简化网表编写。
- 电路可以使用 subprocess 方法使用 Ngspice 库进行仿真，Ngspice 矢量会被转换到 NumPy 数组。
- Ngpsice 库允许 Python 插入电压/电流源。
- 数据分析插件。

在 GitHub 上还有一个同名 PySpice，作者是 Roberto Aguilar。不过这个软件包是 NASA 的天文学星历程序，与 Python 的 SPICE 仿真无关。

4.2.2.5 Python 效果器电路仿真

常见的电吉他特效如下：失真，过载，法兹（Fuzz），合唱，相移，飘忽，压缩，均衡，延时，哇音，混响/回旋，噪声门，变调，和声，颤音，人声，循环，等等。

在基于模拟电路的传统设计中，每个盒子代表一种特效，内置单一功能的电路，分别对应不同的电路设计，比如二极管削波、幅度压扩、BBD 延时线、滤波器等。音频输入/输出用于级联不同特效器，旋钮用于调节电路中的某个参数，踏板让吉他手可以非常方便地用脚来打开特效或直通到下一级。通过串联排列不同特效器，演奏时可以随时切换不同声音特效组合。常见的效果器配置方式如图 4-8 所示。演奏抒情乐曲时，可以使用人声和和声，或者减少特效器使用；演奏情绪爆发的摇滚乐时，通常打开失真和过载来实现感情发泄；还可以使用循环来实现和声。

图 4-8 日本 BOSS 电吉他效果器配置图

来自芬兰的 Henrik Forsten 在 GitHub 上开源了 spice-audio-tools，提供了一个 Python 和 SPICE

来模拟吉他特效电路的仿真过程。该软件包包括以下两个软件。

- wavtospice.py：该工具用于将 WAV 文件转换成 ngspice 能够理解的数值（电压）。
- spicetowav.py：该工具将 ngspice 仿真输出转换成 WAV 文件。

该设计依赖于 Python wave 扩展包。经典的效果器电路设计历史悠久，其大部分都是公开的。虽然现在效果器已经演进到 DSP 架构，可将其别在腰间，小巧、可编程。不过一些吉他手依然偏爱模拟电路构成的效果器。

使用方法如下：

```
$ python wavtospice.py file.wav inputvalues
$ ngspice -b examples/lowpass.cir > spice_output
$ python spicetowav.py spice_output output.wav
```

根据上面推荐的 Python spice 扩展包，加上 Python wave 扩展包及 Henrik 的特效电路仿真，Python 可以很容易地实现效果器电路的原型设计与验证。

实际上，Python 还可以实现更多功能：Python dsptools 或 Audiolazy 可以直接替代特效电路，产生单一或叠加特效，并使用 Twisted UDP 发送给远方的服务器实现直播。读者可以用树莓派来实施这些物联网相关的互动设计。

这也是 Python 参与物联网整个链条进行全栈开发的最佳例子之一，涉及多个环节：信号发生、电路模拟分析、信号处理、传输、传播、娱乐全链条。

4.2.3　数字电路原型设计

由于数字电路容易集成化、规模化，所以数字电路的迭代速度也比模拟电路快，数量多。

4.2.3.1　数字电路输入

数字电路可以采用原理图输入（Schematics Capture）；也可以采用硬件描述语言（HDL）来输入。不过，现在采用硬件描述语言的更多。

4.2.3.2　VHDL 和 Verilog

VHDL 和 Verilog 是 IEEE 标准的数字电子系统设计硬件描述语言。VHDL 由美国军方组织开发，1987 年成为标准。Verilog 作为 Gateway Design Automation 公司（后被 Cadence 公司收购）的私有 IP，1995 年成为标准。相对而言，Verilog 的使用人群更多一些。

4.2.3.3　MyHDL

MyHDL 是一个免费开源的硬件描述和验证的 Python 工具包，官网为 www.myhdl.org。MyHDL 的特性如下。

- 无缝整合：可以将 MyHDL 自动转换为 VHDL 或者 Verilog，然后使用标准工具流程。

- 芯片验证：许多 MyHDL 设计已经在 ASIC 和 FPGA 中进行验证，包括一些高产量的应用。
- 开源设计：MyHDL 是一个开源的纯 Python 包，可以使用 pip 安装，并在 GitHub 上提供源码。

目前 MyHDL 发布了 0.9 版，并支持 Python 3。

由于 Python 是一种高级语言，硬件设计师可以充分利用其特性来为自己的设计进行建模和仿真，并将其作为 VHDL 或 Verilog 的前道流程来使用。

1．建模

Python 的强大和清晰的语法使得 MyHDL 非常适合高层建模。Python 历来以能够处理各种复杂建模问题而著称，在快速应用开发和实验方面也非常突出。

MyHDL 建模想利用 Python 生成器（generator）来解决硬件并发特性建模。生成器是一种可以迭代的函数，而且它会记住上一次返回时在函数体中的位置。所以对于生成器的第二次调用，或第 *n* 次调用时，上次调用所使用的所有局部变量均保持不变。MyHDL 生成器类似于 Verilog 中的 always 功能块及 VHDL 中的 process。

一个硬件模块总是作为一个函数来建模，并返回生成器。这种方式使得它可以很简单、直接地支持许多特性，如任意层次结构、端口关联命名、实例数组、条件实例化等。MyHDL 提供了实现传统硬件描述概念的一些类；同时它支持生成器之间的通信信号类，支持位操作类、枚举类型类等。

2．仿真和验证

MyHDL 内置仿真器运行在 Python 解释器之上，支持在 VCD 文件格式中的波形查看。

MyHDL 还将 Python 的单元测试框架带入了硬件设计。尽管单元测试是非常流行的软件验证技术，但其在硬件设计领域还不是非常普及。如果使用传统 HDL 仿真器进行联合仿真，那么 MyHDL 还可以用作 Verilog 的硬件验证语言。此外，值得一提的是，由于是纯 Python 包，MyHDL 运行于 PyPy 之上，其相比于 CPython 有 14 倍左右的加速，速度相当快。

3．导出 Verilog/VHDL

MyHDL 提供了一个将设计导出到传统 Verilog/VHDL 的工具和流程，包括后期综合和实现在内的路径。其可以转换的子集包括了高层建模和测试，远远超过标准可综合子集。

最后，转换器为 Verilog/VHDL 中的某些艰难任务实现了自动化。例如可以自动处理带符号的算数问题。

4．设计实例

基础的 D 触发器是一种序列型元件，在时钟 clk 的上跳沿发生时，将输入 d 的值传递给输出 q。通常这么简单的电路是不会专门建模的。但是因为触发器非常简单，很适合作为入门的教程使用。

1）器件描述

```
from myhdl import *

def dff(q, d, clk):

    @always(clk.posedge)
    def logic():
        q.next = d

    return logic
```

2）仿真

```
from random import randrange

def test_dff():

    q, d, clk = [Signal(bool(0)) for i in range(3)]

    dff_inst = dff(q, d, clk)

    @always(delay(10))
    def clkgen():
        clk.next = not clk

    @always(clk.negedge)
    def stimulus():
        d.next = randrange(2)

    return dff_inst, clkgen, stimulus

def simulate(timesteps):
    tb = traceSignals(test_dff)
    sim = Simulation(tb)
    sim.run(timesteps)

simulate(2000)
```

函数test_dff创建一个D触发器实例，添加clk信号源和随机激励信号到输入端d。函数simulate用于仿真。在 MyHDL 中，traceSignals 函数用于创建仿真测试台实例，而非直接调用 test_dff。该函数可以创建信号记录文件（vcd 文件），记录仿真阶段所有的信号变化。记录文件可在波形查看器中打开。在小型设计中，这种方式非常有效。

3）仿真波形

GTKWave 是可用于查看 Verilog 仿真的波形数据查看工具，是支持执行 Tcl 脚本和增强的

拖放操作。在图 4-9 中展示的是在 GTKWave 中查看 Verilog VCD 文件的截图。

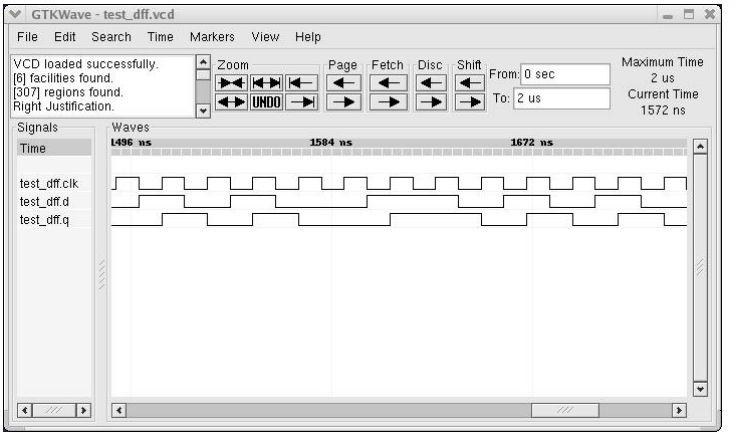


图 4-9　GTKWave 软件使用截图

4）导出到 Verilog

MyHDL 的 convert 函数可以将 MyHDL 转换到 Verilog 代码。

```
def convert():
    q, d, clk = [Signal(bool(0)) for i in range(3)]
    toVerilog(dff, q, d, clk)

convert()
```

运行该代码后，产生 Verilog 代码如下：

```
module dff (
    q,
    d,
    clk
);

output q;
reg q;
input d;
input clk;

always @(posedge clk) begin: _dff_logic
    q <= d;
end

endmodule
```

“千里之行，始于足下”，虽然仅仅是一个最简单的 D 型触发器，但只要积累了足够多的数

字逻辑模型，就可以构成超大规模集成电路。看完这些例子之后，读者可以拿来构建一些简单的耦合逻辑在 FPGA/CPLD 中练练手。

4.3　常见嵌入式微控制器（MCU）

介绍过集成电路本身的设计，我们来看看物联网所需要用到的各类 IC 和原组件。集成电路的分类方式太多，因为本书关注物联网和 Python 编程，所以我们重点介绍 MCU、MPU、通信类 IC、传感器。

4.3.1　MCU 市场状况

在 MCU 市场的前一代产品中，最常见的是 8051。此外，各家私有架构百花齐放。市场占有率较高的有 AVR/PIC12/PIC24/6805/6811/68K/MSP430/H8 等，稍小众的有 PIC32/M8/196/C166/XA 等，每种处理器架构都有自己的侧重点。笔者个人很喜欢收集此类开发板。

市场上对于 MCU 的位数充斥着矛盾的术语，大部分 MCU 称自己为 8/16/32 位架构，有的称自己为 12 位架构。这和 MCU 采用哈佛结构有关，其代码和数据存储器分离，所以两者的宽度可以不同。所谓 12 位架构是其 CPU 指令集的位宽。合理的划分方式应该以寄存器（数据总线）位宽为准。比较典型的有：

- PIC12/16，其指令集宽度为 12/14 位，寄存器是 8 位；
- AVR，其指令集宽度为 16 位，寄存器为 8 位。

所以按照这种划分方式，现在的市场状况如下：

- 4 位架构，大部分需求来自玩具、电子手表和照相机。但大多数厂家，包括中国台湾和日本厂家都已抛弃了 4 位微控制器架构。
- 在 8 位架构中，8051 作为一种工业标准，各家供应商开发过各种标准的（12 个时钟）、增强的（6 个时钟，ROM 到 128KB），甚至兼容 8051 指令而内部架构迥异的 MCU（1 个时钟，可不就是 RISC 嘛），甚至还有运行在 100MHz 的 8051！但继 Intel 退出市场后，Philips/NXP 也转向 ARM。现在 TI/SiliconLab 的某些产品线中还保留着 8051 架构。此外，STM8/AVR/PIC 系列凭着低功耗、低成本的优势占据了一定市场份额。
- 在 16 位架构中，MSP430 是一种比较受欢迎的架构，超低功耗很适合物联网。
- 在 32 位架构中，ARM 的 Cortex-M/A 系列占据了大多数市场，而 MIPS 和其他架构正被边缘化。

接下来，让我们看看由开源硬件所推动的 MCU 平台吧。利用这些平台，开发者可以节省不少开发时间。

4.3.2 Arduino/Wiring

Arduino 是创客圈里最著名的产品。许多人的硬件创新概念由 Arduino 开始。但多数人不了解 Arduino 其实来自 Wiring。

图 4-10 是 Wiring S 的照片。Wiring 平台是 Hernando Barragan 在 Arduion 团队领导人 Massimo Banzi 门下读硕士研究生时所写硕士论文衍生的产品。

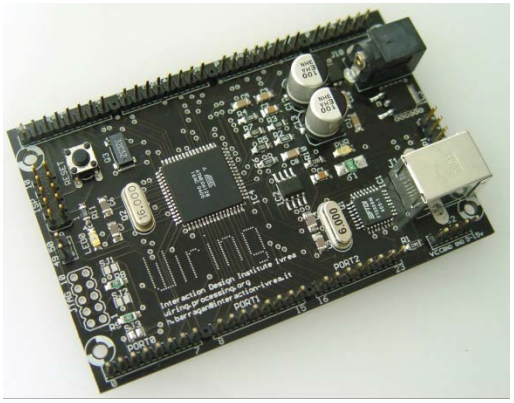


图 4-10　开源 Wiring 平台

在 Wiring 官网上，Hernando 描述了 Wiring 如何从概念，到论文，再到实现，以及其导师 Massimo 和同学如何从 Wiring 分支出 Arduino，并成为一个更加著名的开源设计的故事。具体描述请看本章延伸阅读部分。希望读者能够有空看看，或许从中可以学会如何掌握自己的设计和品牌。

Wring 和 Arduino 的共同特征：

- 简洁的 IDE 开发环境，基于 Processing IDE，可以运行在 Windows/Mac OS X 和 Linux。
- 针对单片机的简单“语言”或编程“框架”。
- 集成完整的工具链，并对用户是透明的。
- 内置 Bootloader 用于加载程序。
- 串行监视器用于检查和数据收发。
- 100%开源软件。
- 基于 Atmel 单片机的开源硬件。
- 大量的在线参考资源，包括命令、库、例子、教程、论坛和项目案例。

Wiring 的 C++ API 在 Arduino 里得到了继承。现在的 Arduino 已经成为品牌，不再受限于 AVR 平台，并开始向 ARM Cortex-M 和 Cortex-A/MIPS/Intel Quark 平台渗透。

4.3.3　ARM mbed

ARM mbed 最初是 ARM 两个员工的个人项目。早期 NXP 和 ARM 的合作很多，所以最初的 mbed 开发板只有一块：NXP LPC1768。很快地，Freescale、STM、Nordic、SiliconLabs、Atmel 都加入进来了。虽然这些厂牌还不能够代表全部品牌，但是光前三位就已经覆盖了大部分的应用市场。

图 4-11 展示的是 ARM mbed 的第一款 LPC1768 mbed 开发板。NXP LPC1768 基于 Cortex-M3，接口丰富，适合物联网开发。

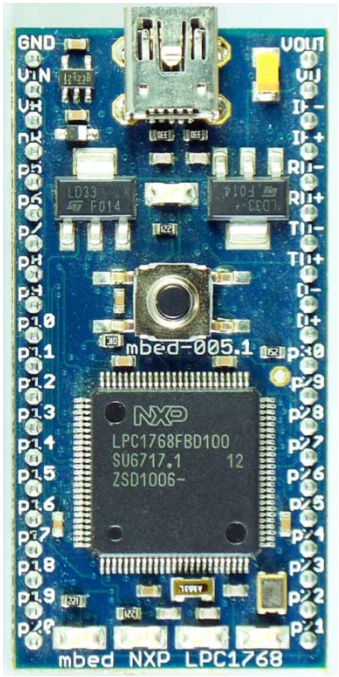


图 4-11　ARM mbed LPC1768 平台

ARM mbed 的技术特点如下。

- 在线 IDE：基于浏览器就可以编写代码，在线编译，下载完成开发。
- 社交化合作开发：在线 IDE 可以导入他人项目，导出自己的项目，实现程序员间的社交化合作。
- 跨硬件平台：跨越品牌和细分架构。同一代码，只要修改 I/O 配置，就可以实现跨品牌、跨 MCU 的配置实施。
- 跨编译器：同一代码，可以在线编译，也可以导出到线下 Keil/IAR/GCC 进行编译。
- 完善的调试支持：提供开源调试器，可以在专业的 Keil/IAR 中进行 JTAG/SWD 调试，

这对于实时系统开发非常重要。

- 完善的中间件和操作系统，构成一个完整的生态。

很快 ARM 收购了 mbed 项目，并在此基础上构建了 mbed OS，成为 ARM 物联网战略的基石。如果拿流行的互联网颠覆论来看，ARM 也颠覆了不少其他行业，包括商业编译器、操作系统、中间件、调试器和私有 ISA 的 MCU。

4.3.4 设计专属架构和专属 MCU

微控制器芯片定制报价是按照逻辑门计价的，而且还有其他费用。但自从低价 FPGA 普及之后，以大量的开源 MCU（软/硬）内核作为参考，许多中小 IC 设计公司都可以设计自己的架构。只需要其指令集是正交的，设计属于自己的微控制器就不再是梦想，并可以在 FPGA 上验证。只不过，后续的 SoC 流片、干净的数据总线、C 编译器、调试接口、市场推广才是需要大家投入资金和精力的地方。

所以，更多的中小公司采用购买 ARM 内核的方式设计自己的 SoC/MCU。连 Freescale 这种曾经的半导体巨无霸、拥有 6805/6811/68000/PowerPC 等诸多 MCU 架构的公司，也专门推出了 Kinetis ARM 内核处理器。同样量级的 TI/NXP/Hitachi/STM，均将支持 ARM 作为公司战略。

2016 年，ARM 被日本 Softbank 收购。出于对 ARM 未来独立性的考虑，或许 MIPS/SPARK 之类的 IP 供应商，以及开源 RISC-V 会迎来一轮新的发展机会。

4.3.5 ARM MCU 差异化竞争

在 ARM 生态链中，不同的芯片供应商之间内核是一样的，那么彼此之间的差异化竞争反而更加激烈。这些厂家必须在片内资源与硅片面积（性价比）、外设与接口（I/O）、供货（二级市场备货）、开发生态（C/C++/操作系统和中间件）的技术支持等各方面展开竞争。早期 ARM 生态链中的厂家还会联合 JTAG 开发工具、C 编译器、操作系统和中间件供应商。但这些开发生态链都被 ARM mbed 一口吃下，全部开源。所以，现在的竞争主要在于性价比、I/O 和供货渠道了。

我们来看看 ARM 阵营中各家供应商的产品线。

4.3.5.1 NXP LPC

在 NXP LPC 系列中，最常用的 LPC1768 是最优产品。其次，LPC175X/6X 的 RAM 配置分别为 16KB/32KB/64KB，现在国内的代理分销价格也不贵，小引脚配置的 LPC812/824 是笔者的最爱品种。LPC54100 是性价比极高的物联网大小核产品。2017 年，NXP 主推的新品主要是 LPC8XX/LPC54XXX 两个系列。

4.3.5.2　Freescale Kinetis

在 Freescale K/KL 系列中，最常用的是 KL25Z128VLK4 和 MK20，RAM 配置为 16KB，ROM 为 128KB，采用 Cortex-M0+内核，批量价格也算公道。

Freescale/NXP 合并为新 NXP 后，据说 LPC/Kinetis 不分彼此均一起推广。但观察实际推广情况，我们发现 Kinetis 的推广力度更大。

4.3.5.3　STM32

在 STM32F103 系列中，最常用的为 C8 后缀，即 20KB+64KB 组合。其次，CB 为 20KB+128KB 组合。其余的还有 10KB/16KB/32KB/48KB/64KB/96KB/128KB 的 RAM 配置，零售市场铺货很多，性价比较高，是很理想的平台。其产品线涵盖 M0/M3/M4/M7，品种非常多。在 ARM MCU 中，STM 无疑是市场占有率最高的厂家。

4.3.5.4　Cypress PSoC

Cypress 的 PSoC 算是小众化的产品，由于其内置可编程的数字模块和模拟模块，因此属于半定制的 MCU。这对于有学术要求和安全特性要求的开发者，值得一试。此外，Cypress MCU 在无线与 USB 市场占据相当多的份额。

4.3.5.5　Atmel ARM 处理器

Atmel 的早期业务包括代工，主要是基于 Flash/EEPROM 的代工。所以其一直支持基于闪存的 8051 和 ARM7TDMI/ARM920/ARM926。但除了 8051，Atmel 一直坚持推广自家 AVR/AVR32 架构。直到最近其才开始增加了对于 Cortex-M 的投入，以求得新的市场份额。2016 年，Microchip 宣布收购 Atmel。这两家的产品线整合后需要仔细观察供货情况。

此外，TI、Nuvoton，及国内的 ARM 处理器也都值得关注。其中，Nuvoton 的推广力度很大，笔者手头就有三块开发板。

4.3.5.6　Python 脚本需要配置更多 RAM

绝大多数微控制器程序都固化在 ROM 中，这也是微控制器程序被称为固件的原因。然而在脚本运行模式下，解释器可以运行在 ROM 中，用户脚本也可以保存在 RAM/ROM 甚至外部 EEPROM 中，但是运行空间却必须在 RAM 空间中展开。所以需要挑选 RAM 相对较大的微控制器，起码为 16KB，如果为 64KB 就比较理想了。不仅仅是 Python，JavaScript/Lua/C#/Java 等运行环境也有类似要求。MCU 的内部布局中 RAM 的硅片面积往往要大于 ROM，所以这决定了大 RAM 的 MCU 比较贵。

4.3.5.7　迷你开发板

笔者个人偏爱 NXP LPC8XX 系列微控制器，并将其做成了一个迷你开发板，用于日常物联

网设计。这种迷你开发板主要采用 C/C++编程，用于各种小型物联网的设备节点使用。配合主机端 Python 脚本可以实现更多功能。比如：

- 信号发生器；
- 红外接收器；
- 树莓派等 Linux SBC 的外设控制；
- WSN RFIC 节点设备或者网关收发模块。

LPC8XX 系列产品特性如下：

- 批量价格低；
- 低价不低能，内置 ARM Cortex-M0+内核，32KB ROM + 8KB RAM；
- 小引脚（Low Pin Count）封装，从 DIP8 到 TSSOP20；
- GPIO 可以通过开关矩阵（Cross-bar）切换；
- 内置 Bootloader 和 SWD 接口；
- 丰富的数字 I/O，包括 USARTx3、SPIx2、I2Cx4；
- 丰富的模拟 I/O，包括 12Msps、12 路、12 位 ADC；
- 特殊定时器，可用于 PWM 电机控制。

与 NXP LPC8XX 类似的产品线还有采用 TSSP20 封装的 STM32F0X0FX 系列产品线。这两家小引脚数 MCU 的共同点是性价比高、焊接容易、支持 ARM mbed 和主流编译器和调试；通信接口丰富，I2C/SPI/UART/USB/CAN 都可以支持，可以承担大量入门级物联网应用设计。

4.4 常见嵌入式处理器和主板

主控 MPU 主要指采用外扩存储器（含 RAM 和 Flash ROM）的微处理器。虽然许多微控制器结构如 8051、80196、68000、51XA、AVR32 之类的都可以使用外扩展存储器，但是笔者还是推荐主流的平台给读者：ARM。

MPU 和 MCU 的开发模式略有不同：MCU 大部分外设都已经内置，工程师使用常见的 EDA 工具就可以独立完成设计。开发 MPU 微处理器需要的硬件设计和软件开发门槛比较高。硬件方面必须使用多层板，需要针对动辄几百 MHz 至 2GHz 频率的存储器总线进行微带设计等；软件方面需要交叉编译器和 Linux 内核、驱动开发能力及 Android 系统适配等。所以，一般应用层面的开发者往往使用 SoM（System on Module，国内也称为核心板）系统模块，并根据模块接口制作应用开发板，以增加需要的外部设备或者适配模具外形。

综上所述，本节会将 MPU 与对应的主板或核心板一起介绍。

4.4.1　ARM 架构

在嵌入式领域，ARM 架构是绝对的王者。属于 MPU 范畴的有 ARM9/ARM11/Cortex-A5/7/8/9 以及 A15/53 等大小核处理器。以这些内核为主，许多半导体大厂推出了一系列 MPU/SoC。由于其技术门槛较高，技术支持资源也非常有限，因此，其中一部分厂家配合第三方合作伙伴提供了完整的主板。例如：

- Broadcom ARM11 树莓派（RapsberryPi，简称 RPi）；
- TI OMAP BeagleBoard/BeagleBone。

4.4.1.1　树莓派

树莓派（RaspberryPi，简称 RPi）3，采用 Broadcom 4 核 Cortex-A7 64 位处理器。由于树莓派价格低廉，性价比非常高，体型小巧，自称“卡片电脑”，所以在创客市场非常流行。其最新版本是树莓派 3，如图 4-12 所示。针对树莓派的各类设计和外设层出不穷，它是读者学习 Linux 的最佳平台之一。

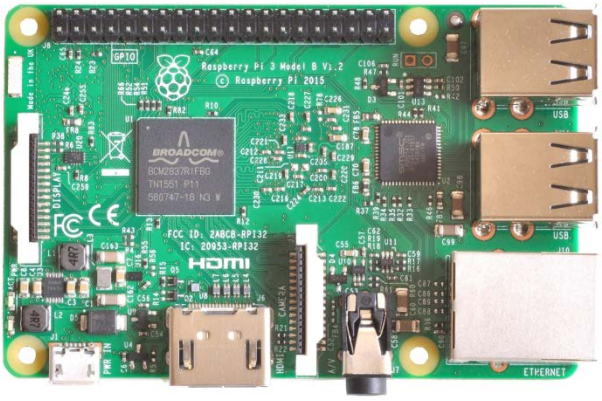


图 4-12　树莓派 3

目前树莓派 3 运行的 Linux 如下：

- Raspbian，针对 RPi 的 Debian；
- Ubuntu Mate；
- Pidora，针对 RPi 的 Fedora；
- ArchLinux，轻量级 Linux；
- RISC OS，一款 RTOS；
- Raspbmc，HTPC 媒体中心；
- OpenElec，HTPC 媒体中心；

- Windows 10 IoT；
- Chromium OS；
- OpenWRT；
- Android。

4.4.1.2 BeagleBone Black

BeagleBone Black（简称 BBB）是 BeagleBoard.org 推出的低成本 Linux 单板机，采用德州仪器的 Cortex-A8，其外形如图 4-13 所示，可以使用相当多的操作系统：

- Angstrom Linux（默认）；
- Debian Linux；
- Ubuntu Linux；
- Arch Linux；
- Gentoo；
- Minix；
- RISC OS；
- Android。

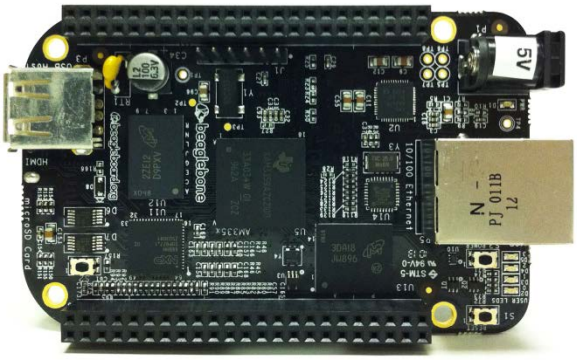


图 4-13 BeagleBone Black

与树莓派相比，BBB 的 I/O 要多许多，种类也更多：

- 3 组 I2C 总线；
- 1 组 CAN 总线；
- 1 组 SPI 总线；
- 4 组定时器；
- 5 组 UART 串口；

- 65 个 GPIO；
- 8 组 PWM 输出；
- 7 组模拟输入（1.8V 12 位 ADC）。

作为工业和通用目的的 BBB，其提供了更多的 I/O，可以接入更多种类的传感器，非常适合开发商业化产品。

说起来，Broadcom ARM11 和 TI OMAP 都是退出手机应用处理器竞争后的剩余产品，但是其定位不同，这直接决定了其后续的生命周期。现在让我们对比一下这两者，各项对比列在表 4-1 中。

表 4-1　树莓派与 BeagleBone Black 对比表

对比项目	树莓派 2	BeagleBone Black	胜出者
处理器	ARM11/Cortex-A7 Quad	Cortex-A8	BBB
图形处理	Videocore 1920x1200	1440x900	树莓派
操作系统	Linux	Linux	BBB
功耗	350mA	460mA	树莓派
GPIO	8	65	BBB
视频输入	CSI	N/A	树莓派
扩展板	较少	较多	BBB
复制性	需授权	基本开源	BBB
社区支持	很强	强	树莓派
入门价格	<USD35	USD45	树莓派

注意　表 4-1 仅仅对比了树莓派和 BeagleBone Black。随着更新版本的树莓派、Board-XM 的上市，某些对比项目会有变化。

BBB 使用了更新型号的处理器和更多的 I/O 扩展。在 CPU 方面，其被树莓派 3 反超。总的来说，树莓派在视频 I/O 上力压 BBB，而 BBB 在处理能力和硬件扩展上要远胜树莓派。

1．BBB 适合的项目

BBB 适合的项目如下：

- 连接大量传感器的项目。因为 BBB 具备更多扩展口。
- 打算商用的项目。开源特性使得 BBB 构建最小系统相对容易。
- 即插即用。BBB 内置 eMMC，所以可以节省启动配置时间。

2．树莓派适合的项目

树莓派适合的项目如下：

- 多媒体项目。树莓派的 Videocore 和 CSI 为视频输出和输入提供了很好的支持，同时

模拟视频和音频也提供了更多选项。但 Videocore 是闭源项目。

- 社区驱动项目。如果项目更多需要社区投入精力，那么树莓派可以吸引更多注意力，而且许多项目可以直接使用。
- 具备 GUI 的项目。这也和树莓派在视频图形方面的优势有关。

根据笔者的观察，BeagleBone/BeagleBoard 面对的是较为专业的创客和行业人士，而树莓派一开始就以教育界为目标，所以价格定位更低些，普及率较高。

4.4.2 其余的 ARM Linux 主板

ARM 的许可证模式让 Fabless 设计公司站到了和国际半导体大厂同样的高度。借助大中华地区的半导体产能和广大的国内外消费市场，中国国内的 ARM 芯片迭代甚至超过了国际大厂，购买 ARM 许可证也比大厂积极。

在中国，消费类电子产品的竞争非常激烈。诸多公司在高清机顶盒、高清电视机和平板电脑市场上提供了各种方案。物联网时代，智能硬件、可穿戴设备、AR/VR 的出现，使得各个新创公司的开发比传统厂商更加积极。在面对极客的小众市场上还活跃着以 Allwinner/Rockchip/Intel 为主的三大流派开发板。

- 以 Rockchip 为主的 MK802 系列，外形类似 U 盘，采用 USB 供电，HDMI 输出视频，使用 Ubuntu/Android。
- 以 Allwinner（全志科技）为主的各类中国派，如 PCduino、BananaPi、WaxberryPi、OrangePi 等，使用树莓派兼容的 Raspbian/Ubuntu/Android。
- 第三种力量即 Intel 平台开始出现，外形类似 MK802 风格以及平板和工控板平台，使用 Ubuntu/Android/Windows 8/Windows 10（Intel 在平板市场也在尝试学习 MTK 的交钥匙白牌战略）。

在表 4-2 中列出了部分国内市场常见的 ARM Linux 主板。

表 4-2 ARM Linux 主板

名　称	品　牌	CPU	RAM（MB）	ROM	接　口	OS（操作系统）
RaspberryPi Zero	Adafruit	ARM1136	512	N/A	SoM	Raspian
BeagleBoard-XM	Beagle	A8	512	4GB	ETH	多种 Linux
BeagleBoard x15	Beagle	A15 Dual	2000	4GB	ETH	多种 Linux
PandaBoard ES	PandaBoard	A9×2+DSP	1024	N/A	Wi-Fi/BLE/ETH	多种 Linux，Android
MarsBoard	HAOYU	A7×2	1024	8GB eMMC	ETH，HDMI	Linux
MK802	Rikomagic	A8	512/1024	4GB	Wi-Fi	Android 4.0，Ubuntu
MK808	Rikomagic	A9×2	1024	4GB	Wi-Fi	Android 4.1，Ubuntu

续表

名　　称	品　　牌	CPU	RAM（MB）	ROM	接　　口	OS（操作系统）
PCDuino	Linksprite	A8	1024	2GB	ETH，HDMI	Ubuntu
CubieBoard	Cubietech	A7	1024			Debian，Android 4.0
CubieBoard 2	Cubietech	A7×2	1024			Debian，Android 4.0
NanoPi	FriendlyARM	ARM926EJ	64	N/A	Wi-Fi/BLE	Debian Jessie
NanoPi2	FriendlyARM	A9×4	1024	N/A	Wi-Fi/BLE	Debian Jessie，Android
NanoPi2 Fire	FriendlyARM	A9×4	1024	N/A	ETH	Debian Jessie，Android
BananaPi	Elastos.org	A7×2	1024	N/A	ETH	Android 4.4，Debian，Ubuntu，Raspbian，Cubieboard
BananaPi M3		A7×8	2048	8GB eMMC	ETH+Wi-Fi，SATA	Android5.1，Debian，Ubuntu，Raspbian
OrangePi		A7×4	512	N/A	ETH，USB，CSI+HDMI	Android 4.4，Debian，Ubuntu
AWorks	ZLG	ARM926EJ	64	128MB	ETH	Embedded Linux

注意

1. 以上表格仅作为参考，具体规格请参考主板供应商官网最新信息进行核实。

2. 大部分 ARM 嵌入式 MPU 主板都具备 GPIO（含 UART、SPI、I2C）及 USB，所以不再单独列出。

3. 零售市场价格波动较大，上述主板在本书编写之际的价格区间为 49~360 元人民币，这与具体 BOM 配置有关。

从目前来看，在这么多眼花缭乱的主板中，Allwinner 方案最多，Rockchip 次之，再次才是 TI/Broadcom/Samsung 的处理器。国内几家针对开源市场的供应商如友善之臂、PCduino、CubieTech、OrangePi、BananaPi、OrangePi 等都在 Allwinner 的 A10/A20/H3/H8 处理器上根据不同的配置进行了外设周边多种组合，外扩 GPIO 也保持与树莓派的兼容性，所以其对于创客和普通消费者很有吸引力。

其中“价格杀手”Orange Pi PC，报价 99 元人民币。虽然其缺乏板载 Wi-Fi/BLE，但是 4 核 A8+1GB RAM+SATA 接口，已足够吸引人了。在本书编写之际，笔者发现友善之臂的 Allwinner H3 开发板居然只有 69 元人民币。

在这些平台上，往往可以运行 Linux，相当部分的高配主板甚至可以运行 Android。其 Python 的使用方式与树莓派和 BeagleBoard 类似。如果国产 SBC 质量保持稳定，并加强技术支持，包括硬件驱动、Bootloader 等可以保持开源，则凭着巨大的产能和超高性价比，其可以是许多产品和工程的首选平台。

廉价 Linux SBC 往往采用的也是针对消费市场的 MPU，物美价廉，可以在大多数家用和商用环境中使用。而在一些恶劣环境，如极端温度、湿度、粉尘和电磁噪声环境中，其可能会出现问题。所以针对这些场景应用时，用户需要做些高低温和电池兼容实验，以确定是否能够应用。另外一种选择就是找到具备这方面应用背景的主板供应商，如研华之类专门提供工业主板的品牌供应商。

4.4.3 MIPS 开发板

中国的龙芯指令集兼容 MIPS，龙芯 1C100 开发板可以使用 Arduino/Microduino Shield 扩展板。从图 4-14 中可以看到 PCB 中部上下两排的 Arduino 扩展插座，以及右下角的 Microduino 扩展插座。

图 4-14　国产龙芯 1C100 智龙开发板

国内除了 ARM 生态链伙伴，MIPS 内核的 Linux SBC 也不少，包括：

- 龙芯 SoC 开发板；
- OpenWRT 系列开发板，Boardcom/Atheros 等；
- 高清播放器开发板（非 Android+ARM 平台）。

对于开发者来说，技术成熟度、供货渠道、生态和社区支持不可缺少。虽然 MIPS 可以使用 Linux，但是 Android 对于 MIPS 的支持是不足的，尤其是高清盒子进入 Android 时代后，许多厂商如 Realtek（瑞昱）直接放弃了 MIPS 平台。

MIPS 加油！

4.4.4　x86 mini-ITX

我们在关注 ARM+Linux 组合的同时，还可以留意入门级的 x86 主板。尤其是那些 mini-ITX 主板以及瘦客户机，其是嵌入式应用的理想配置。甚至 Intel 都推出了 NUC（Next Unit of Computing）来迎合这方面的需求。

mini-ITX 是由 VIA（威盛电子）定义和推出的一种结构紧凑的微型化主板设计规范，目前已被各家厂商广泛应用于各种商业和工业应用中。它是用于小空间、小尺寸的专业计算机。在电商市场中检索“mini-ITX”、“HTPC”、“广告机”就可以找到一大堆。图 4-15 是 Gigabyte 出品的一款面向 DIY 市场的 mini-ITX 主板。

图 4-15　Gigabyte mini-ITX 凤凰板

这些主板的成本并不高，软件兼容性却非常高，可以安装各类操作系统，硬件接口也非常丰富：

- 传统的 PC 主板接口如 PCI、miniPCIe、SATA。
- 足够多的 USB（x4）和串口（x6），适合物联网扩展。
- Ethernet 可用于各种物联网连接。
- VGA/HDMI/DP/Audio 接口也都具备，可用于多媒体交互。

和 ARM SBC 相比，属于 PC 类别的 mini-ITX 主板的缺陷如下：

- 功耗较高。
- 缺乏 CSI/DSI 原生视频 I/O，需要 USB/HDMI 桥接。
- 缺乏原生 SPI/I2C/ADC/PWM 等嵌入式 I/O，需要 UART/USB/miniPCIe 桥接。

这些主板很适合那些对于功耗要求不高的场合，如软路由器、POS 机、物联网网关、IP 电话网关等。采用小容量 SSD 盘替代硬盘后，可以进一步减少系统功耗。相对而言，ARM SBC 虽然在功耗上胜过 x86 SBC，但是实现同样的扩展 I/O、处理能力的 SBC，其价格也往往是 x86 SBC 的好几倍。没有统一的外形规格、缺乏互换性也是 ARM SBC 的短板之一。

4.5 常见传感器和执行器

移动终端，尤其是智能手机的快速普及，带动了传感器，尤其是 MEMS 微机电传感器的发展，短期内就降低了成本。在一台智能手机中，很有可能具备多种传感器。常见的传感器如表 4-3 所列。

表 4-3 智能手机内置传感器

名　称	简　介	原理或常见型号
加速度计	测量 *X*/*Y*/*Z* 三个轴向的加速度和方向	MPU6050
陀螺仪	测量三个轴向的角速度和旋转了多少角度	ITG3205
电子罗盘	测量平面中的磁场强度和方向	HMC5883
开关感应	监测组件间的组合动作	霍尔元件
光线	测量光照以调整屏幕亮度	光敏三极管或二极管
气压	测量气压补充 GPS 参数以及天气监测	Bosch
温度	测量设备内部的温度升降	处理器片内或 I2C 总线外接传感器
距离感应	感应人脸的接近并开关屏幕和触摸屏幕	红外收发器件
重力感应	感应重力方向以切换屏幕	采用加速度计实现重力感应
计步	单独的计步传感器较少	可使用加速度计替代
心率	监测心率	采集 LED 照射后反射红光变化可以推算心率
血氧	监测血氧	采集红外和红光反射率之比可以推算血氧
指纹	用于手机授权管理	采用电容测量法的指纹传感器已经开始普及
辐射	较为少见	日式手机常见
拾音器	读取声音	从电容式改成 MEMS
摄像头	拍摄外部影像	CMOS 摄像头可以作为心率、血氧和其他智能算法的基础
紫外线	获取环境中的紫外线强度	采用特殊光谱的光敏器件
湿度	获取环境中的湿度百分比	湿度敏感的阻容器件，DHT11
触摸	手机中常见的触摸屏模块	多数从单点电阻式转向多点电容触摸技术
GPS	获得设备当前位置、方位角、速度和时间	GNSS 模块非常多
NFC	通过线圈感应 RFID Tag	电池耦合和负载调制，Mifare

在所列传感器中，陀螺仪是非常重要的传感器。早期陀螺仪异常昂贵，主要用于军事（如导弹）和车辆惯性导航。日本游戏厂商将陀螺仪引入了游戏领域，降低了成本。随后，大量采用 MEMS 工艺的陀螺仪进入了手机领域。由于早期陀螺仪需求仅限于游戏，出于成本压力，许多低成本手机没有内置陀螺仪。VR 播放器依赖陀螺仪提供 3D 空间角度和角速度参数，而 VR

风潮让许多手机用户发现，原来自己的手机里根本没有配备陀螺仪传感器！

4.5.1　虚拟传感器

某些“传感器”如重力、方向、线性加速度、旋转矢量，属于加速度计和陀螺仪进行某种算法（滤波）计算后产生的虚拟传感器。

随着各类传感器融合的趋势愈加明显，将传感器提供的原始物理量进行计算，得出间接物理量后提供给高层应用的虚拟传感器会越来越多。除了上述例子，车辆的路面摩擦力、飞行空速等根据建模计算的传感器应该归入虚拟传感器。

4.5.2　智能传感器

与直接采集物理量的传感器如温度、湿度、加速度和陀螺仪相比，有些传感器有内置的处理器单元，自成一个子系统，其往往被称为“智能传感器”。GPS 就是典型的例子。

现在的传感器迭代往往不再是简单地采集物理量，而是新的传感器配合内置智能算法的组合。

- 全景摄像：采用多（鱼眼）镜头组合成全景摄像，这也是目前最热的“VR 直播”。
- 3D 摄像：采用双目或多目镜头组合成 3D 影像。
- VR 摄像：全景其实不算是 VR，而 VR 摄像是利用摄像头+红色激光点+红外摄像头，拍摄点云后实时合成 3D 模型。
- 主动降噪拾音器：通过两组或多组拾音器采集环境声音，利用算法将环境声音中需要提取的声音进行加强，或对噪声进行智能识别。
- Wi-Fi/BLE：利用人体在无线网络中对于网络信号的衰减来获取位置信息。

通过对于传统传感器彼此之间的融合，以及与各种新算法的结合，业界正在尝试各种各样的新传感器应用。在独立的 VR 设备、平衡车（Segway）等智能设备中已经采用了上述传感器。许多传感器比如 NFC/无线充电等，目前虽然使用目的单一，但是其日后一定会发掘出新型应用。

在 Sparkfun/Intel 设计的模块中，使用了 Intel Edison 扩展板和 Invensense 的 9DOF MEMS 传感器，内部由三轴加速度计、三轴陀螺仪和三轴磁力计组合而成。其外形如图 4-16 所示。市场上更常见的 IMU（Inertial Measurement Unit，惯性测量单元），多采用带浮点计算单元的 ARM MCU，通过传感器融合算法，输出计算后的姿态物理量。此类内置微处理器和算法的传感器，可以归入“智能传感器”。一些半导体原厂如 Freescale 提供此类模块。

图 4-16　Sparkfun 和 Intel 合作设计的 9DOF（Degree of Freedom，自由度）模块

在消费场景中，除了汽车的传感器比较多，无人机也算是相对复杂的系统，其会采用更多的子系统：飞控、数传、云台、VR 直播，并会在物体追踪和智能避障等中采用许多“智能传感器”。

4.5.3　专用传感器

离开五彩缤纷的消费场景，其他行业的传感器则追求稳定、精确。不过其价格也就更加昂贵了。传统传感器的分类方法有许多，但是其主要按照被测物理量划分，或者按照传感器的工作原理划分。

1．物理量划分法

按照物理量，可以分为湿度、温度、压力、位移、流量、液位、受力、加速度、转矩传感器等。

2．工作原理划分法

按照工作原理，可以分为以下传感器。

- 电学传感器：电阻式，电容式，电感式，磁电式，电涡流式。
- 磁学传感器：铁磁物质的物理效应。
- 光电传感器：光电效应和光学原理。
- 电势传感器：热电效应、光电效应和霍尔效应。
- 电荷传感器：压电效应。
- 半导体传感器：压阻效应、内光电效应、磁电和气体变化。
- 谐振传感器：利用机械参数改变谐振频率的原理，测量压力。
- 电化学传感器：以离子导电为基础，用于分析气体、液体、酸碱度、电导率等。
- 生物传感器：由生物分子识别部分和转换部分组合，是生物、化学、物理、医学、电子多方面融合的高新技术。

在传统工业、农业、军事等相关物联网领域，其传感器的种类和销售额是非常惊人的。笔

者正在汇总农业应用中使用的各类传感器并设计一些专业设备，希望能够将传感器、控制器、处理器和云计算方案一起推荐给广大农场人员。传感器部分包括：

- 土壤温度、湿度传感器；
- 土壤盐碱度传感器；
- 光照传感器和光合辐射传感器；
- 二氧化碳传感器；
- （水）氨氮传感器；
- （水）钾离子传感器；
- pH 值传感器；
- 水温传感器；
- 水位传感器；
- 雨量液位传感器；
- 灌溉系统流量传感器；
- 气象站系统（环境温度、湿度、气压、风力、风向）等。

但就目前来看，农用传感器的价格短期内无法下降太多。对于许多农业生产者来说，他们从事的是一种“靠天吃饭”的高风险行业，但利用传感器和物联网技术却可以最大限度地规避农业生产中的损失。

4.5.4　执行器

执行器是自动控制系统中的执行机构和控制阀组合体。它在自动控制系统中的作用是接收来自调节器发出的信号，以其在工艺管路的位置和特性，调节工艺介质的流量，从而将被控设备控制在生产过程所要求的范围内。

执行器按所用驱动能源分为气动、电动和液压执行器三种。而 MCU/MPU 对外输出控制信号的手段是电压、电流或者工业总线。接口部分是电机驱动和电液伺服阀等。工农业生产领域的执行器属于自动化控制领域，涉及机械、机电、液压、气动等学科，本书不做介绍。下面仅列举一些常见的执行器。

表 4-4 列出了常见的“执行器”种类与品牌。如果将“执行器”引申到更加宽泛的概念上，那么所有受控设备和系统都可以归类到“执行器”。比如 LED、门禁继电器和电机、扬声器等。

表 4-4　常见的执行器

名　　称	简　　介	品　　牌
继电器	电压控制开关，子分类有电磁式、感应式、电子式、热效应式、气动式、电动机式	施耐德、欧姆龙、宏发、松川、德力西、ABB 等

续表

名　　称	简　　介	品　　牌
电机	驱动各类气动和电液设备，子分类有高温电机、调速电机、直线电机、交流与直流电机、伺服电机、步进电机、同步与异步电机、变频电机等	三菱、三洋、横河、ABB、松下、西门子、日精，以日本品牌居多
电磁阀	电磁控制流体的自动化基础元件。用在工业控制系统中调整介质的方向、流量、速度和其他参数	SMC、Parker 等
流量伺服阀	接收电气模拟信号后，相应地输出调制的流量和压力。其既是电液转换元件，也是功率放大元件，能够将小功率微弱电气输入信号转换为大功率液压能（流量和压力）输出	Parker、MOOG、Vickers、Rexroth 等
水泵	机械能转换成液体能量，子分类：往复泵、柱塞泵、活塞泵、隔膜泵、转子泵、螺杆泵、液环泵、齿轮泵、滑片泵、罗茨泵、滚柱泵、凸轮泵、蠕动泵、扰性泵、叶片泵、离心泵、轴流泵、混流泵、漩涡泵、射流泵、喷射泵、水锤泵、真空泵、旋壳泵、软管泵、蜗杆泵	格兰富、威乐、ITT、KSB、Pentair、Ebara……

4.6　物联网通信集成电路

物联网设计中的连接性集成电路或通信模块是重点关注对象，所以笔者专门在第 5 章阐述此类 IC 的开发。

无论是系统组网还是联网，在所有此类集成电路与主控 MCU/MPU 的连接接口中，高速接口作为片内外设（On-Chip Peripheral）直接接入内部总线（AHB/APB），如 CAN、以太网等，可以通过文件或套接字系统服务编程。除此之外，这基本上都可以归类到几类串口，可以通过 SPI/I2C/UART/USB 等方式接入系统控制器。本节简单列举了一些常见的通信类 IC，具体参阅表 4-5。

表 4-5　常见的通信类 IC

种　　类	简　　介	常见型号	接　　口
NFC	近距离电磁耦合传输	RC522	I2C/UART/SPI
Bluetooth 3.0	经典蓝牙	CSR8670	UART/SPI/USB
BLE	蓝牙低功耗	nRF18122/DA14580/CC2540	UART/SPI
Zigbee	IEEE 802.15.4 Mesh 网络	CC2530/JN5168	UART/SPI
6LowPAN	IEEE 802.15.4 中的 IPv6 网络	CC2530	UART/SPI
SubGHz	低于 1GHz 的私有无线协议	CC1100/CC1200/SI4432	SPI

续表

种　　类	简　　介	常见型号	接　　口
IEEE 802.3	以太网	ENC28J60/W5200/DM9000	UART/SPI/GPIO
IEEE 802.11	Wi-Fi	ESP8266/CC3200/MTK7681	UART/SPI
GPRS	时分多址蜂窝数据	MG2639	UART/USB
CDMA	码分多址蜂窝数据	MC2716	UART/USB
LTE	4G 蜂窝数据	ME3860	USB/UART
LoRa	低功耗长距离传感器网络	SX127X	SPI

注意　表 4-5 不作为采购参考依据，读者需要自行在市场上寻找芯片或模块。

4.7　嵌入式系统开发语言演进

Python 与 C/C++的关系密切。CPython 是各大系统中的默认配置，Python 默认的解释器是用 C 写的。本章主要介绍嵌入式开发环节中 C/C++的使用现状。

C 语言是嵌入式系统中的优势语言。除汇编语言之外，C 语言是最贴近裸机的编程语言。C 几乎主导着嵌入式系统从 Bootloader、固件、驱动程序到操作系统和应用程序的每个环节。随着物联网应用中大量复杂中间件的导入，C++紧随其后已经进入了深嵌入式领域。

PyMite 和 MicroPython 这两种嵌入式 Python 主要依靠 GCC 交叉编译器进行平台移植。而 Embedded Linux 中也是采用将 CPython 交叉编译而来支持 Python 的。针对 ARM/MIPS 平台的完整版 Python，则通过各自本地编译器进行编译。所以，离开 C/C++，几乎没有 Embedded Python。

4.7.1　从汇编到嵌入式 C

从汇编到 C 语言编程，是解放生产力的一大步！ 其对程序员的意义不亚于洗衣机、燃气灶对于家庭主妇的革命意义。

软件代码最初的形式是打洞机，实现二进制输入，后来出现了终端输入代码。甚至笔者学习计算机原理课程的教学设备，也是 8051 配合键盘+LED 的原始终端；输入 0～F 按键，等价于 MCU/MPU 的 HEX 机器码。我们可以将其与汇编机器码画上等号。

汇编程序，尤其在早期的 4/8 位微控制器时代的电子行业发挥了很大的作用。大量软件，无论是家电、工业控制，还是游戏行业（如红白机），均采用汇编程序编写。但是随着软件复杂度的上升，其维护成本过于高昂，迭代速度也很缓慢。所以，最接近机器原始特性的 C 语言开始在嵌入式开发领域得到普遍采用。在小规模应用中，采用优化过的 C 编译器生产的代码不比人工代码多出多少，随着项目的复杂度上升，C 编译器产生的代码反而比人工代码要少。MCU

片内资源愈加丰富，C 编译器的优势越发明显。

彼时，MCU 的架构百家争鸣，还没有被 ARM 一统天下。所以嵌入式 C 语言供应商需要配合半导体供应商以及仿真器开发商进行不断优化和升级，否则容易出现兼容性问题；且其售价非常昂贵。当时的开发生态链条非常恶劣。

C 不是一开始就支持嵌入式，尤其是资源受限的嵌入式 MCU 需要定制的嵌入式 C 语言。8051 内置支持位寻址，片内 RAM 寻址空间有限，这些都是针对汇编优化的结构。许多从计算机专业毕业的同事一开始在 8051 上编程时，对于底层架构不熟悉，其变量很快就把有限的片内 RAM 用完了。这时，他们不得不将所有变量重新修改一次。所以，ANSI C 和嵌入式 C 还是有许多的不同点，需要开发者深入了解。

使用 8051 支持 C 语言编程是一个重要的“里程碑”。这里必须提及 Keil 公司，Keil 创始时是一家德国公司。Keil 的英语发音：/ki:l/，即“基尔”；德语发音：/kaıl/，即“凯尔”。该公司的 C51 编译器针对 8051 结构做了许多优化，如：

- 针对布尔类型的变量采用 sbit/bit 类型；
- 采用相对跳转替代长跳转指令，以节省代码空间；
- 指定变量存储空间以充分利用 8051 架构；
- 定义大中小类型的存储模式；
- 以及其他优化项。

因为产生代码最小、支持的芯片供应商最多的技术优势，所以 Keil 占据了 8051 C 编译器的主导地位。Keil 与另外一家编译器公司 IAR 分享了嵌入式 C/C++语言中商业编译器的大部分市场份额。

基于 GCC 的嵌入式编译器既有商业版也有免费开源版，包括 ARM 公司本身也维护一个版本：ARM GCC Embedded。各个架构的 MCU/MPU 几乎都有对应的 GCC 编译器。这为开源硬件的开展奠定了基础。

必须使用汇编语言的场合

C 语言是嵌入式行业主要的编程语言。实际上没多少人喜欢使用汇编语言，但在某些场合下人们却不得不采用汇编语言：

- 底层或原子操作，如临界区进出等，需要使用内联汇编。
- 需要精确定时的操作，如精确到一个机器周期的操作 NOP。
- 需要减少存储器空间使用，精确定位的操作。
- 需要高度人工优化的操作，如 DSP 核心算法。即使 DSP 已经开始使用 C++，许多核心算法优化依然依靠汇编。
- 特别低级的架构，没有 C 编译器支持。
- 特别新的架构，处于原型开发，还没有 C 编译器支持。

4.7.2　从 C 到 C++

与汇编语言相比，C 语言大大提升了程序编码效率。随着应用的日益复杂，需要复用原有的设计，支持面向对象编程的 C++可以进一步加快软件研发速度。C 语言是现有嵌入式软件的主流语言。半导体供应商都交付 C 语言库。随着应用的日益复杂，C++已经开始进入嵌入式系统。常见的平台有 GCC、Wiring、Arduino、Maple 以及 mbed。

首先，GCC 符合 ANSI C 标准，同时还支持 C++。据笔者了解，AVR 是 GCC 支持的第一款 8 位处理器。最初使用 AVR-GCC 开发的时候，笔者曾经怀疑 C++是否会占用太多资源。试用后发现，AVR-GCC 的代码效率并不低！和 Atmel 的 Debugger 配合也很完善。

在 AVR-GCC 编译器和库函数的基础上，Wiring 工程推出了 Wiring API。这是一套完整的嵌入式系统 C++接口库。现在 Intel Quark 的 Wiring x86 也可以追溯到这个 API 库。Arduino 在 Wiring 基础上，定义了 Arduino 的 Shield 扩展板，诞生了一个可以硬件扩展的生态。并且由于这个生态，吸引了大量的开发人群。以至于现在许多半导体供应商提供开发板，都通过兼容 Arduino 的 Shield 扩展以充分利用这个开发生态链。

Wiring C++ API 是 Arduino 的软件开发基础，也标志着 C++在深嵌入式行业的成功应用。接下来，ARM mbed 提供的也是 C++ API。这标志着主流厂家已经开始将 C++导入嵌入式平台。

C 和 C++是两种设计哲学的语言，分别对应于面向过程（Procedure Oriented）和面向对象（Object Oriented）。早期的计算机和现在的嵌入式系统有着类似的问题：资源受限。所以，当时设计中最重要的就是要节省计算资源，并提升性能。随着计算资源成本的下降，软件系统越发复杂，开发周期和可维护性的重要性压过了性能要求。

我们在经典的计算机书籍中，总是看到这一点被反复提及：OOP 实际上是软件工程发展到一定阶段，为满足软件质量和开发周期要求而出现的。OOP 的经典特征如下：继承，多态，封装。许多教科书上讲述 OOP 时过于枯燥，笔者找到一篇网文：《汉语是世界上唯一一种面向对象的高级语言》，其行文有趣，推荐一读，可以帮助读者理解 OOP。

4.7.3　压缩 C++的系统消耗

为何 C++可以在资源受限的处理器上使用？道理很简单，嵌入式 C++往往只是 C++的子集。同时，现在的处理器能够提供的计算资源也与前几代的处理器不可同日而语。

笔者曾经为 Freescale 的 KL 系列提供 C++适配支持并移植 Arduino 核心库。如果未经优化，这会出现很大的系统开销。在 C 语言中，除了 malloc 等函数，大多数函数仅使用堆栈。在 C++中，创建对象无论如何都会用到堆（Heap），这时内存管理非常重要。C++类的构建、析构和访问都比 C 要耗费宝贵的 RAM 和 ROM 空间。

在机械工业里，Baremetal 指的是没有加工过的金属原料。在嵌入式开发中，Baremetal 术

语被借用，特指没有操作系统的 MCU/MPU 系统，中文应该翻译为“裸机”。

Quantum Leaps 公司 CTO Miro Samek 在 embedded.com 网站上有一篇文章 *Building Bare-Metal ARM Systems with GNU*（《用 GNU 开发 ARM 裸机系统》）值得一读。原文网址请参见本章延伸阅读部分。该文的第四部分指出开发 MCU 裸机系统过程中使用 GNU C++需要注意的编译选项。现在简要介绍如下。

C++需要一些额外的初始化步骤来调用构造（Constructor）方法，而 GNU C++还要产生一些额外的段（Section）来保存构造和析构（Destructor）方法表格。所以链接脚本需要链接这些段，startup 代码需要调用这些构造方法。

如果没有仔细挑选编译器选项，而使用了标准 C++的设置，C++的消耗（overhead）可以非常容易地到达 50KB 以上，这会使得开发陷入困境。这也使得 C++无法适用于 LPC8XX/STM32F030 等低端 MCU。但如果限制 C++的特性，则 C++的这些系统冲击可以减少到最少。该文中讲解了如何将 GNU C++产生的代码缩减至最少，最终优化结果仅比传统 C 语言代码多出 300 字节。所以，低端 MCU，包括 AVR/Arduino 以及 LPC8XX 也可以充分使用 C++带来的收益。

```
CPPFLAGS = -gdwarf-2 -c -mcpu=$(ARM_CPU) -mthumb-interwork \
-mlong-call -ffunction-sections -O \
-fno-rtti -fno-excetptions –Wall
```

从上面的代码中可以看到，makefile 中至少要关闭两个编译开关。

- -fno-rtti 关闭了关于 C++运行时所需要的类型识别功能（如动态类型转化），即不再产生每个带虚函数的类的信息。关闭该编译开关可以减少若干 KB 的代码生成。
- -fno-exceptions 关闭了异常处理中的某些代码，可以减少若干 KB 的代码生成。

仅仅关闭这两个开关可能还不足以减少足够的 C++消耗。或许还需要在其他方面如 new/delete/malloc/free 函数以及堆管理等处进行优化，这样才能够将 C++适配到低端处理器。根据 Quantum Leaps 的指引，笔者采用 ARM-GCC Embedded 编译器编译了 Kinetis MCU 的许多库和演示代码。

4.7.4 C++适合物联网开发

AVR-GCC，同时支持 C 与 C++语言编程，但是 C++依然比 C 语言更消耗资源。Arduino 为何要使用 C++呢？现在我们回顾 Arduino 的历史，可以知道 Arduino 的开发者团队熟悉 Java，开发了 Processing，所以采用同样面向对象编程的 C++是比较顺理成章的。

相信许多工程师还是会追问：有必要用 C++吗？毕竟 C++的实施在系统开销上是要大于 C

的。国内团队甚至还推出过 Arduino/Wiring API 的兼容 C 语言函数库，宣传重点是开销比 C++要小。

从发展角度来看，大型软件系统使用 C++（甚至包括本书推荐的 Python）是完全必要的。C++的系统开销，尤其是 ROM 开销不值一提，主要是 RAM 开销需要控制一下。现在的 MCU 存储器容量都比 8 位机时代要大许多，可以使用 C++。笔者在 NXP 的 LPC812 上大量使用过 mbed C++代码，C++开销在优化的情况下完全可以接受。

解决可能性之后，OOP 的必要性是什么？C++究竟在多大程度上能够推动物联网和嵌入式固件的开发呢？

首先，复杂软件系统的软件质量要求使然。C 和 C++语言是两种设计哲学的编程语言，分别为面向过程和面向对象。对于开发者来说，OOP 提供了一个更加抽象的编程思想，可以加快复杂项目的开发速度。虽然 C 也可以按照 OOP 的设计思路去编程，但 C++提供了更加直接的语法支持。

其次，C/C++兼容性使然。虽然耗费了一定的存储器和运行性能，C/C++天然的兼容性使得程序在两者间切换并不复杂。笔者就借鉴了 Arduino 的 ADB USB 函数库，并成功移植到 Kinetis FRDM-KL25Z 开发板上。ARM mbed 也是高层使用 C++，底层依然大量使用供应商提供的 C 语言库。

最后，系统多态性使然。物联网连接技术复杂多变，这种硬件的多态性，更适用于 OOP 的设计思路。

4.7.4.1　物联网连接技术的多态性

关于物联网的多态性，笔者在 M2M 的 MODEM 开发中体会颇深。该项目来自运营商需求，需要同时支持 CDMA 和 GSM 两种移动通信模块，模块均来自中兴物联网，但是其芯片平台分别来自高通和 MTK。这导致其核心 AT 指令集类似，但是却有所不同。相信这个情况适用于大多数供应商，因为基础的 AT 指令集都是类似的，而模块之间、网络之间、品牌之间必然有差异。让我们看看 ZTE CDMA/GSM 的异同点。

主叫拨号指令是不一样的：

```
CDMA：AT+CDV10001\r\n
GSM：ATD10086;\r\n
```

被叫接听是一样的：

```
CDMA & GSM：ATA
```

在 C 语言里的源码基本上都是：

```
zte_gsm.c / zte_gsm.h
void gsm_dial(char * number);
```

```
void gsm_answer(void);

zte_cdma.c / zte_cdma.h
void cdma_dial(char * number);
void cdma_answer(void);
```

由于两个文件源码中分别针对两种不同串口，实施两种不同的函数，所以在两个 C 语言模块中，两套 AT 指令集存在相当程度的代码重复。在高层调用中一定如下所示：

```
gsm_dial("10086");
cdma_dial("10001");
```

如果工程师偷点儿懒，还会进行代码间的复制/粘贴，然后就可能会出现问题。这是典型的硬件多态性的例子。如何求同存异呢？

笔者在该项目中的 C++源码则是这样的：

```
modem.cpp / modem.h
void modem::dial(char * number); // Virtual function
void modem::answer(); // send "ATA" to serial port

gsm.cpp / gsm.h
class gsm(modem);
void gsm::dial(char * number);

cdma.cpp / cdma.h
class cdma(modem);
void cdma::dial(char * number);
```

基类是 modem 类，gsm 和 cdma 分别是子类。而且 modem::dial()是虚函数，可以被两个子类各自实现，而 modem::answer()可以被子类继承使用。这样如果 AT 指令集两者一致，则直接放到 modem 类中；如果不同，则分别放在两个不同类中。高层调用代码如下：

```
gsm.dial("10086");
cdma.dial("10001");
```

总的来说，目标码反而节省了 ROM。由于串口驱动封装在 MODEM 中，源码中仅需要一个串口中断服务程序。C++多串口调试在 Debugger 中是一件麻烦事，因为多个串口在 Debugger 调试时显示为同一个断点！解决方式是分别调试，确保底层驱动完美后联调。

在物联网中，此类现象大量存在。尤其是在多串口应用中，除了蜂窝数据 MODEM，还有串口 Wi-Fi、串口 BT/BLE、串口 Zigbee、串口红外、串口 6LowPAN 等。所以在驱动层面使用 C++的面向对象思路可以在一定程度上简化物联网设计，简化代码。需要完整代码的读者可以参考 ARM mbed 网站中 XBee 模块的官方代码。

4.7.4.2 C++更适合复杂系统

与 C 语言相比，C++在维护性、扩展性、软件质量和开发速度方面更占优势，在复杂软件

系统中这一点尤为突出。如果仅仅控制 GPIO/SPI/I2C/ADC/PWM 这些底层硬件，只需要 C 语言的话，那么物联网中更多的是各类中间件：通信堆栈、GUI、文件系统、特殊硬件编程等。这些往往需要利用 C++来编程。最典型的还是系统应用和 GUI 了。虽然 Linux 的主要界面之一 GNOME 采用的 GTK+是 C 语言在 GUI 中的代表，但是 C++ GUI 的选择更多。

- Qt，一种跨平台的 GUI，使用 C++。
- Fox/FLTK/wxWidgets，此类 GUI 都使用 C++。

虽然 C++的许多特性争议较大，但是这并不影响 C++在嵌入式领域的渗透。

4.7.4.3　纯 C 语言构建 OOP 编程

当读者习惯于 OOP 编程思想并爱上这种思想时，就会主动使用 C++来开发软件。但是许多情况下，系统只提供 C 语言开发环境。虽然 C++从语法层次支持 OOP，但凭着 C 语言也是可以构建出 OOP 编程思想的代码的。要实现 OOP 的代码，需要大量使用 C 语言中的指针和结构体 struct。但是 C 语言不支持 C++的许多特性，如函数重载、包管理机制等。所以，C 实现的 OOP 还是与 C++有所区别的。

person.h：

```
typedef struct person{
    int age;
    char* name;
}PersonClass;
```

student.h：

```
typedef struct student{
    PersonClass p;
    int math;
    int english;
    void (*setName)(char*, PersonClass*);
    void (*setAge)(int, PersonClass*);
    void (*setMath)(int, PersonClass*);
    void (*setEnglish)(int, PeronClass*);
}StudentClass;

void setName(char* name, PersonClass* p){
    p.name = name;
}

void setMath(int score, PersonClass* p){
    p.math = score
}

void setEnglish(int score, PersonClass* p){
```

```
    p.english = score
}
```

main.c：

```
#include "stdio.h"
#include "person.h"
#include "student.h"

main(){
    PersonClass me;
    me.name = "allankliu";
    me.age = 30;
    printf("%s is %i year old",me.name, me.age);

    StudentClass you;
    you.setName = &setName;
    you.setName("Kirin",&(you.p));
    you.setAge = &setAge;
    you.setAge(100,&(you.p));
}
```

此段代码参考了“顽客”的博文《OOP 类与继承》，网址请查看本章延伸阅读部分。在 C 语言中引入面向对象等更加抽象的设计方法和模式，不仅仅是广大程序员的想法。受到各类高级语言的启发，爱丁堡大学博士 Daniel Holden 新开发的 C 语言标准库 libCello，就支持一系列高阶特性：

- 接口，改善代码的设计。
- 异常，控制错误处理。
- duck 类型，泛型函数。
- 构造器和析构函数。
- 各类语法糖。

4.8 C/C++的编程模式和技巧

在最初的MCU应用中，往往没有操作系统。裸机上的程序就是一个主循环＋中断服务（ISR）。主循环调度也被称为 round robin 方式。

即使是简单的程序，也需要使用一种编程技巧：回调函数（callback）。可以这么说，能够熟练使用回调函数和函数指针可以反映出程序员对于 C 语言的熟练程度。

这些所谓的技巧在相对复杂的 Windows/Linux 开发中很常见。开发 MCU 程序则相对较少，这一方面和 MCU 开发语言的演进有关，另外一方面和开发者的学习经历有关。所以，我们有

必要总结一下嵌入式系统中常见的设计模式和一些技巧。

4.8.1　C/C++设计模式

各种高级设计模式，使得开发更加简洁、快捷。嵌入式设计模式涉及硬件访问、并发和资源管理、状态机、安全和可靠性四方面的设计模式。

4.8.1.1　硬件访问模式

- 硬件代理模式（Hardware Proxy）：将硬件访问封装在类或结构体中，并独立于硬件实现。
- 硬件适配器模式（Hardware Adapter）：创建的适配器组件将不同参数需求与供给接口对接。
- 中介模式：采用外部组件来协调不同组件之间的复杂交互和配合行动。
- 观察者模式：也被称为 Pub/Sub（发布/订阅）模式，用户高效地分发数据，如传感器数据。
- 去抖动模式：抑制间歇硬件信号，用于过滤键盘抖动之类的硬件毛刺时间。
- 中断模式：处理紧急的异步事件信号，用于硬件中断处理等异步事件，这是最常见的编程模式。
- 轮询模式：定期检查新的硬件信号，这也是最常见的编程模式。

4.8.1.2　并发和资源管理模式

- 循环执行模式：在无限循环中调度，这是最简单、常见的模式，体现公平获得资源。
- 静态优先级模式：通过预先设定不变的优先级进行调度，在公平基础上加快了紧急事件的响应。
- 临界区模式：通过禁止任务转换（或中断）保护资源，避免竞争，这是通信相关最常见的模式。
- 守卫调用模式：通过互斥信号保护资源，其缺点是会导致优先级倒置。
- 队列模式：通过消息队列 FIFO 模型序列化访问资源，这是任务间或进程间异步通信的常见模式。
- 汇合模式：协调复杂任务同步的模式。
- 同时锁定模式：同时锁定资源以避免死锁。
- 排序锁定模式：以特定顺序锁定资源以避免死锁。

4.8.1.3　状态机

- 单事件接收器模式：单一事件接收器解决同步和异步事件消息传递。
- 多事件接收器模式：多个事件接收器解决同步事件，这是最常见的实现。

- 状态表模式：采用二维表在存储状态转换信息，其扩展容易，但不容易处理嵌套。
- 状态设计模式：常见状态对象实现状态机。
- 分解与状态模式：分解复合状态实现逻辑与状态。

4.8.1.4 安全和可靠性

- 二进制反码模式：逐一添加原始数据按位取反来确认通道的数据完整性，在检测通信通道和存储器失效方面很有效。
- CRC 模式：通过冗余校验码来确认数据完整性，能够检测出比 CRC 本身长度更长的错误。
- 智能数据模式：可简单理解为数据对象与方法，用方法校验对象合法性，实现运行时检查。
- 通道模式：将传感器数据进行转换，提供大量的冗余单元。
- 保护单通道模式：在单通道内添加数据和检查点验证来提升通道模式，提供轻量级冗余。
- 双通道模式：创建多个（同构和异构）通道识别错误和失效，并有选择地容忍一定错误的发生并继续提供服务。

以上设计模式在许多语言和系统中都可以实现，也在不少设计书籍中介绍过。只不过，嵌入式系统编程语言从汇编、C/C++到 Java，不同层级的开发应用模式都有所不同。我们可以在日常开发中仔细体会这些设计模式的具体好处。

4.8.2 回调函数

回调函数是通过函数指针调用的函数。如果把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，就是回调函数。回调函数在特定的事件或条件发生时，由另外的一方调用，用于对该事件或条件进行响应。

函数 f()的指针 fp，被作为参数交给调用者 c，这个过程是注册。而当事件 e 发生时，由 c 通过 fp 来调用 f()，这个过程为回调。f()就是回调函数。随着物联网引入网络堆栈的日益增多，回调的使用日渐频繁。要理解回调，先看看我们最常见的编程模式：同步编程模式。

4.8.2.1 同步编程模式

通信协议都是分层设计的。一般来说，函数调用是从高层逐层调用到底层函数，这种调用往往是同步调用。高层函数必须等待底层函数逐级返回才能够继续运行下一个函数，所以同步调用也是堵塞式的调用。

拿生活中的例子来说，一个顾客要退货，打电话给客服专线。顾客是高层函数，“要退货”是数据，客服人员是底层函数。这里要根据高层需求做处理。同步方式就是“电话不要挂，请

稍等，进音乐……”，从整个系统来看，顾客+电话+客服都在等待事件处理完毕，这样其他顾客就得不到服务了。

读者这时候就会说：为什么顾客不把自己的联系电话留给客服人员，事件处理完毕后由客服通知顾客呢？这样顾客和客服资源可以分离，只有需要的时候才连接。这就是现实生活中的回调。

一个封装完美的软件，应该和外部解耦。这就好比顾客和客服彼此就应该是陌生人，而不是依赖个人关系才能够得到服务的道理是一样的。

再看物联网开发中的情况。相当多的代码库使用固化的函数调用，即同步模式来实现。甚至许多蜂窝数据通信模块供应商提供的参考代码也是这样设计的。同步堵塞型代码非常容易识别，检查代码中 delay()函数的使用数量就可以做出判断。同步模式代码会带来一系列问题：

- 底层通信协议栈将接收的数据转发给应用程序应该采用什么方式？
- 如果要将底层通信协议打包成库，是否需要包含高层应用的函数头文件？
- 如果应用层没有实现这个代码，通信协议访问一个空的高层应用代码是否会跑飞？
- 如果多个高层函数访问同一个底层函数，那么底层函数如何判断高层调用者的来源和业务逻辑？
- 设计如何实现模块化和清晰的层次？

在通信分层设计中，高层函数将自己的通知函数指针注册到底层函数中，当底层函数接收到数据后，通过函数指针通知高层函数处理。这个完整的回调函数可以比较优雅地解决这些编程中的矛盾和问题，实现不同进程间的异步通信。虽然例子中会以高层函数和底层函数为例，但实际上回调与彼此的逻辑关系无关。

4.8.2.2　异步编程模式

从字面上理解，所谓回调（callback）可以简单地理解为底层函数往回调用高层函数（call backward）。其隐含的意义是将调用者和被调用者分离解耦，实现消息响应、事件处理的异步通信。回调是实现异步编程的重要方式。具体的实现方式如下：

（1）将高层用户函数注册为回调函数，如 registerCallback(application_logic)；

（2）底层函数接收到数据后，调用之前注册的回调函数，如 triggerCallback()→application_logic()。

在 C 语言中，回调函数采用函数指针实现。在面向对象的设计中，则采用回调类实现。在其他高层应用设计中，回调函数往往是两个进程间异步通信的手段。这两个进程可以是运行在不同处理器或者服务器中的进程，也可以是同一系统内的底层和高层任务进程。

回调的好处如下：

- 解决通信协议和应用层之间的异步通信。

- 底层通信协议可以提供灵活的可重用接口，可以打包成库。
- 底层协议不需要固化配置高层的业务逻辑。

gsm.h：

```
void gsm::registerCallback(void (*fptr)(void), CallbackType type);
```

gsm.cpp：

```
#include "gsm.h"

void registerCallback(void (*fptr)(void), CallbackType type) {
    if (fptr) {
        _callbacks[type].attach(fptr);
    }
}
```

main.cpp：

```
#include "gsm.h"
static void setup()
{
    gsm.registerCallback(gsmRing, GSM::CbRING);
}

main(){
    if (gsm.gsmRing != NULL){
        gsm.gsmRing();
    }
}
```

读者可以仔细观察各类开源设计中的 USB、TCP/IP 和 M2M MODEM 相关源码，它们都是采用回调函数来实现的。回调函数实现的异步通信的编程模式，也被称为消息响应模式。在实际工程设计中，许多供应商提供参考代码以加快用户的开发。读者可以先看看其通信代码是同步模式还是用回调实现的异步编程模式。如果是同步堵塞编程模式，也没有使用 RTOS 进行任务调度，那么供应商提供的代码只有寄存器设置是有意义的，其余部分基本就需要读者自己升级改造了。

4.8.2.3 回调的其他用途

除了通信协议栈，点对点通信中还会用到回调函数。回调函数还可以用于：

- 中断服务例程；
- 协议解码，如 XML 解码。

虽然大多数嵌入式系统中没有进程和线程的概念，但是却有主循环和中断服务例程。这非常类似于主线程和后台线程。而线程间通信可以采用回调函数。主循环将中断服务例程作为回

调，硬件中断时会调用该例程。至于协议解码，无论是红外解码中的位解码，还是在 XML 流中接收到某个标签，都会调用事先注册的对应回调函数。所以，回调函数的使用范围很广，需要读者仔细掌握。

4.8.3　有限状态机模型

有限状态机（FSM，Finite State Machine）是一个数学概念。它既可以用于硬件设计，如计数器、串行奇偶校验、UART 收发器等；也可以用于软件设计。

NXP 的 LPC8XX/LPC18XX 和 LPC43XX 芯片中有一种特殊的外部设备：可配置状态定时器（SCT，State-Configurable Timer）。其可用于各种定时、计数、输出调制和输入捕获操作，如计数器、函数发生器、PWM 电机控制（最多四路）、红外遥控编码、解码、PWM 解码、PPM 调制等。一般软件 FSM 会利用状态变量来控制状态转换，而 SCT 则将这些状态变量以硬件寄存器的形式实现。图 4-17 中展示了 SCT 的配置与工作原理。LPC43XX 支持 32 个状态寄存器。状态寄存器越多，越可以实现更加复杂的状态机。利用 SCT 配合 FSM 设计，可简化软件 FSM 的编写或者减少中断次数，提高系统整体运行速度，并提供不同外设组合与功能扩展。不过，这要求开发者充分理解 FSM 设计模式和 SCT 硬件。

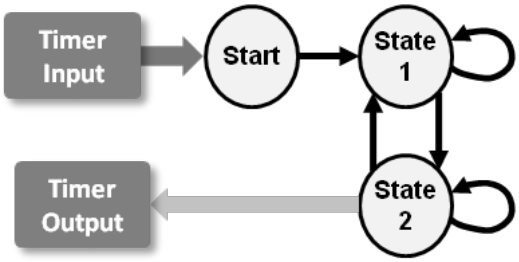


图 4-17　NXP LPC MCU 中的 SCT 配置示意图

FSM 的基本实现思路是采用一张表保持所有状态。每个状态有当前状态、进入条件，以及执行的动作和离开条件。FSM 在软件中的实现方式大致是 switch/case、if/elif/else 或者函数指针数组。这与实现异步事件驱动的实现方式类似。两者的区别如下：

- FSM 判断的对象是状态变量；
- 事件驱动判断的对象是异步事件。

以 MP3 媒体播放器为例，其状态有 POWDOWN、STANDBY、MP3、RADIO、RECORD；按键有 POWER、MODE、PLAY/PAUSE、UP、DOWN、LEFT 和 RIGHT。请查看以下伪代码。

define.h：

```
enum button{POWER,MODE,PLAY,UP,DOWN,LEFT,RIGHT};
```

```
enum state{POWDOWN,STANDBY,MP3,RADIO,RECORD};
```

FSM.c：

```
switch(state){
    case POWDOWN:
        if (button == POWER)
            state = STANDBY;
        break;
    case STANDBY:
        if (button == POWER)
            state = STANDBY;
        break;
    case MP3:
        if (button == POWER)
            state = POWDOWN;
        else if (button == MODE)
            state = RADIO;
        else if (button == PLAY)
            state = STANDBY;
        break;
    case RADIO:
        if (button == POWER)
            state = POWDOWN;
        else if (button == MODE)
            state = RECORD;
        else if (button == PLAY)
            state = STANDBY;
        break;
    case RECORD:
        if (button == POWER)
            state = POWDOWN;
        else if (button == MODE)
            state = MP3;
        else if (button == PLAY)
            state = STANDBY;
        break;
    default:
        break;
}
```

Event.c：

```
switch(button){
    case POWER:
        if (state == POWDOWN)
            state = STANDBY;
        else
            state = POWDOWN;
```

```
            break;
        case MODE:
            if (state == MP3)
                state = RADIO;
            else if(state == RADIO)
                state = RECORD;
            else if(state == RECORD)
                state = MP3;
            break;
        case PLAY:
            if ((state == MP3)||(state == RADIO)||(state == RECORD))
                state = STANDBY;
            else if(state == STANDBY)
                state = MP3
            break;
        case UP:
            volumeUp();
            break;
        case DOWN:
            volumeDown();
            break;
        case LEFT:
            next();
        case RIGHT:
            previous();
        default:
            break;
    }
```

除了以上的 switch...case 结构外，还可以使用 state[i]()等形式构成里外两层判断结构。开发者可以自行确定使用哪种方式。

在实际设计中，FSM 还要分不同的层次：Hierarchy FSM（HFSM），即分层状态机。将状态分类，抽离出来，将同类型的状态分别作为一个状态机，并使用一个大的状态机来维护这些子状态机。基于 HFSM，还有行为树（BT，Behavior Tree）和有向无环图（DAG，Directed Acyclic Graph）设计。

在嵌入式软件设计中最重要的就是有限状态机的设计。其可以在 UI、驱动、协议等各个方面存在。由于嵌入式软件往往没有操作系统，所以适当地分层实现分层有限状态机可以简化程序的设计。

4.8.4　善用结构体

结构体（struct）是由基本数据类型构成并用一个标识符来命名的各种变量的组合。结构体在 C 语言中的作用非常大，但许多程序员没有好好利用结构体的便利性。

结构体是一个新的自定义数据类型，因此结构体变量也可以像其他类型的变量一样进行赋值、运算，不同的是结构体变量以成员作为基本变量。结构体还可以有结构体数组和结构体指针。在一个结构体成员中可以包含另外一个结构体，这被称为嵌套结构体。

4.8.4.1 封装对象

如果想要在 C 语言中实现 OOP，结构体是唯一的选择。在前面的代码中：

```
typedef struct person{
    int age;
    char* name;
}PersonClass;

typedef struct student{
    PersonClass p;
    int math;
    int english;
    void (*setName)(char*, PersonClass*);
    void (*setAge)(int, PersonClass*);
    void (*setMath)(int, PersonClass*);
    void (*setEnglish)(int, PeronClass*);
}StudentClass;
```

这里不仅仅是封装了类的属性，还封装了方法。结构体是新的数据类型，可以理解为新的类，初始化后的结构体变量则是创建的对象。如果不使用结构体类型，那么必然使用 student[]、math[]、english[]等数组来保存，这无法有效、安全地封装数据。

4.8.4.2 通信协议解码

使用枚举定义报文字段的偏移量来进行通信协议解码也是可行的。但是 Python 的 struct 模块提醒了笔者：在 C/C++中使用 struct 结构体或许是更好的选择。

```
typedef struct {
    unsigned int Header;
    unsigned char Datalen;
    unsigned char ProtocolVersion;
    unsigned char DataType;
    unsigned char *DataBuf;
    unsigned int Crc;
    unsigned int Tail;
}STXETXClass;

unsigned char buf[256];
struct STXETXClass packet;

memcpy(packet,buffer,5); // Received 5 bytes
unsigned char ds = packet->Datalen;
```

```
memcpy(&packet.DataBuf,buffer+5,ds-2+2);
```

以上代码仅用于说明利用结构体来进行报文解析的方法，读者需要自行修改，适配到自己的编译器和具体项目中去。

4.8.4.3　复杂参数传递

一般来说，C 语言函数的参数是通过寄存器或者堆栈进行传递的。在 8051 中，函数调用中传递的第一个参数，如果是 unsigned char，会放在 R7 寄存器；第二个参数，如果是 unsigned char，会放在 R6 寄存器。更多参数通过压入堆栈，进入子函数后将参数弹出堆栈后赋值给局部变量。不同的 MCU/SoC 的方式是类似的，差别在位数和寄存器的使用上。所以，C 语言的参数受限于堆栈和内存空间。大部分嵌入式 C 语言会在编程使用文档中明确提示，在何种存储模式下，将采用何种方式传递参数：通过寄存器或者堆栈。

在大多数情况下，函数参数的数量不会很多。但也有比较极端的情况，举一个典型的场景：GSM 短消息收发。它需要传递的参数有目标号码、短消息中心号码、语言编码、发送编码、有效时间、内容缓存区、长度等。

```
sms::sendsms(unsigend char* target, unsigned char* smsc, unsigned char language, \
    unsigned char encoding, unsigned int expire, unsigend char* databuf, \
    unsigned int len);
```

在笔者编写的代码中，最多传输过 9 个参数！这种设计，会将整个参数通过堆栈传送一次。这等于将所有参数复制到堆栈，然后在子函数中复制到局部变量。好浪费资源！这很容易引起堆栈溢出。

正确的做法是定义一个短消息发送结构体，以上的参数都是结构体的成员，将结构体的指针作为参数传递。这样压到堆栈里的只是指针而已，可以通过寄存器传递，速度远比压栈快，同时避免了堆栈溢出的风险。

4.8.4.4　多参数传回

在 Python 中我们经常可以看到以下这种例子：

```
def myfunc(name, age, sex, english, chinese, math, physics, chemical):
    learning_everything()
    return teather, father, mother, address

sir, man, woman, location = myfunc("kirin",6,"girl",100,100,100,100,100)
```

Python 神奇的地方不在于它可以介绍多少输入参数，而是它可以同时返回多个值！ Python 返回的也不是多个参数，而是一个 tuple，包含了多个返回值。

C/C++理论上无法返回多个值，但是可以参考 Python 的实现。还是以 GSM 的短消息解码场景为例，需要从输入缓存区中读取原始字符串，返回的数值应该有解码后的字符串、语言编

码、语种、对方号码、时间戳等。我们可以定义一个结构体用于短消息解码：

```
gsm_sms_strcut myfunction(unsigned char arg1)
```

gsm_sms_strcut 其实是一个自定义结构体类型，需要返回的数值是结构体成员。C 使用结构体 struct、C++使用对象作为返回值，就可以一次返回多个值。只不过，Python 在语言特性上就支持了多数值返回，而 C/C++则需要单独定义。

除了采用结构体的返回方式，C/C++语言还有一种返回参数的方法：把输入参数作为返回值。具体做法如下：

```
unsigned int myfunction(unsigned char arg1, unsigned int arg2,\
     unsigned char *arg3, unsigned int *arg4)
```

其中 arg3、arg4 是两个指向 unsigned char 类型的指针，它们是输入参数，不过修改它们指向的变量（或对象）后返回函数，那么这些参数实际上就是返回值。return 语句采用堆栈返回结果，而结构体采用指针返回结果，参数类型必须是指针。

4.8.5 C/C++协程

笔者之前一直对嵌入式系统开发中 delay/delay_ms/sleep 之类的函数耿耿于怀。尤其是学了 Python，发现 Python 的 yield 可以用于实现异步协程，又对比了 Java/C#中的 yield 之后，就想找到方法替代这些函数。因为在许多软件中，delay 就是本地空循环，原地打转。如果本地延时等待时间超过秒级，则往往是需要交出控制权的时候了。

习惯了硬件、习惯了汇编级别的开发，都会从硬件角度考虑问题。return 通过堆栈将控制权（即程序指针）交还给调用者。不使用堆栈，那么它们如何交出控制权呢？那么 yield 是如何在物理机器上实现的呢？yield 其实是协作式的多任务，任务 A 通过 yield 将控制权交给任务 B，任务 B 在适当的时候将控制权交给 C，C 再交给 A。如此循环。既然不是通过堆栈这种纯粹硬件的方式实现的，那必然是通过操作系统、解释器或者汇编级别的技巧实现的。

在众多程序员的努力下，原生不支持协程的 C/C++可以通过宏定义和库函数来实现。但这有各种限制，比如依赖于硬件和操作系统等。以上内容具体可以参考本章延伸阅读中的相关内容。

在后面提到的 mbed RTOS 中，提供了 Thread::wait 方法。一旦调用这个方法，RTOS 调度器会把当前线程置于等待状态，允许其他线程运行。不过这已经不是协程，而是线程切换了。

C/C++语言处于开发的底层，依赖于具体的硬件实现。在深嵌入式领域，互联网的持续交付、快速迭代、敏捷开发几乎无法实现。切换一次平台的代价不小，所以我们需要依赖于一种开发生态。通过硬件的抽象与虚拟化，以及平台与开发者的合力，实现一定程度的加速开发。

4.9　开发生态选择

本节主要介绍 ARM 生态链中的若干重要环节：开发工具（C 编译器，仿真器，仿真环境）和软件（含操作系统和中间件）。

4.9.1　工业标准与厂家私有指令集架构

ISA（Instruction Set Architecture）即指令集架构。不同品牌半导体公司均可以获得许可实施的指令集架构都是工业标准架构。与此对应，只有单一品牌推出并采用的为私有指令集架构。在 MCU 的黄金时代，除了当时的工业标准 8051/68000 架构，不同的半导体供应商还提供了多种架构，可谓百花齐放。ARM 一开始并不占优。但是它的这种 IP 授权业务模式配合 IC Foundry 成为现在的主流模式，成为嵌入式市场上最常见的工业标准指令集架构。

虽说市场环境出现了标准化、同质化趋势，但是笔者依然推荐大家对各类架构保持一定的技术敏感。比如：

- 异构平台，大小端或者 ARM Cortex-A 配合 Cortex-M；
- Cypress PSoC 架构，虽然 PSoC 采用了 ARM Cortex-M0+做内核，但可编程的硬件模块却很吸引人；
- FPGA 的软核，虽然 ARM 免费公开了其 M0+的可综合软核以及 M1，但是 FPGA 供应商还同时提供针对自己产品优化的软核 IP；
- 多核架构，如 XMOS 和 Tensilica 等，以及 GPU 计算；
- TI 的 MSP430，在低功耗上做了很多优化；
- STM 的 M8 内核，可能是市场上最便宜的 8 位 MCU 之一。

4.9.2　硬件与软件平台选择

开源硬件 Arduino 特别喜欢 Atmel 品牌和 AVR 架构。Arduino 最近的升级版开始采用 Atmel 的 ARM 处理器及 Intel 的 Quark。与 Arduino 相比，专业开发者应该首选 ARM mbed，因为它适合开发商业产品。原因如下。

第一，多样性：Arduino 虽然支持 AVR/ARM 两种架构，但其主要选择 Atmel MCU 做平台，而 mbed 提供了更多厂家的选择。因为 ARM 的特殊业务模式，以及主导了大部分嵌入式 MCU/MPU 市场，许多 SoC 都是采用 ARM 内核 MCU 开发的，所以 mbed 的 MCU 多样性完胜 Arduino。Arduino 虽然也有 AVR/ARM/Quark 架构，但是其特点是私有架构。

第二，中间件与协议堆栈：ARM 将 mbed 作为物联网平台，其收购了许多公司并且都往这个篮子里装，包括 mbed OS、mbed cloud server、6LowPAN 等。其提供的 IP 和协议堆栈的复杂

度与完整度要大幅度超越 Arduino。

第三，行业趋势：早期 Arduino 大量采用 5V 供电 DIP 封装的 AVR，以至于大量创客依旧集中在这个平台上，这不符合主流的行业趋势：即 3V3 甚至更低，SMT 芯片封装。不过这一点现在已经有所改变，其现在大多提供 3V 供电、QFP/QFN 封装版本。

第四，调试手段：调试支持是非常重要的，在相当多的情况下，printf()只适用于高层的非实时调试要求，无法满足实时调试以及 HardFault 等故障排除。AVR/ARM 处理器支持 GCC/GDB 的 JTAG 调试，但是 Arduino IDE 的 Processing IDE，却切割了这个调试平台，使得调试困难。Maple 项目虽然也使用 Wiring C++ API，但是其卖点之一是支持传统编译与调试。ARM mbed 可以在多种编译器环境下支持 JTAG/SWD 调试。

第五，代码重用：GitHub 和 mbed 看重社交开发模式的目的是代码重用和构建开发团队，有共同兴趣的人群很容易一起推动项目迭代。

第六，货源充分：ARM Cortex-M/A 的货源实在是太多了。

所以在实际工程开发中使用 ARM mbed 会是一个比较合适的选择。

4.9.3 编译器选择

mbed 做了一件了不起的事情，解决了项目和编译器依赖的问题。同样的代码可以自由地在不同编译器间迁移。笔者日常使用的工具链是这样的：ARM MCU，尽量使用 mbed 支持的 ARM 型号，同时使用 GCC 和 Keil 两种编译器。

被 ARM 收购后，Keil 的 UV4 成为 ARM 的主流编译器。GCC 是一大类编译器。这里特指的是 mbed 推荐的 ARM 官方维护的 ARM GCC Embedded 和 Codesourcery 的 GCC。

因为 Keil + JTAG/SWD 调试比 GCC + GDB + JTAG/SWD 要简单直接，笔者开发底层应用的时候非常依赖于 Keil 的调试能力。尤其是遇到 HardFault 之类严重问题的时候，必须依赖调试手段来搞清楚究竟是哪一段代码导致的运行错误。

Keil 免费评估版本支持 32KB 编译，所以小型项目可以直接使用。如果项目日渐复杂，则可以使用 GCC 回避这个 32KB 限制问题。一旦完成内核和驱动调试后，高层应用整合可以大胆地使用 GCC 编译。使用 GCC 的代价是 ROM/RAM 会占用更多，但是符合开源项目特性。C++ 本身就是强类型语言，许多问题在编译阶段就可以排除了，需要注意的是存储器的使用所导致的运行时问题。

如果读者要问，大型系统也遇到 HardFault 怎么办？如果一个系统已经复杂到一定程度，笔者建议采购一套正版的 Keil 编译器配合正版（教育版）的 J-Link。这是一笔合理的投资。

4.10　常见操作系统

和桌面的操作系统不同，嵌入式操作系统基本上都是实时操作系统（RTOS，Real Time Operation System），具备在足够快的确定时间内处理事件并做出响应的能力。桌面操作系统并不具备实时特性，所以与安全相关的设计必须使用 RTOS 或者某种实时调度算法。

4.10.1　无操作系统

嵌入式程序没有 RTOS 也可以开发程序。笔者最初接触到嵌入式编程是从 8051 开始的。最典型的就是主循环配合中断服务程序（ISR）构成整个系统。相比之下，更加低级的 PIC12 连中断源也没有。这些比较原始的 MCU，既没有 C 语言，也没有中断，完全依靠程序轮询去实现一些简单的应用。

现在产品的开发成本（开发人工和周期）和上市成本（渠道成本）早就超过了 BOM 成本。入门级的 ARM Cortex-M0 批量价格均只有 0.30 美元。由于入门级开发工具基本上都是免费的，而且厂家提供的 BSP 日趋标准化，因此基于 Cortex-M 系列控制器的嵌入式系统开发周期可以很短，开发成本大大降低。无论是开发者还是项目经理，都不愿意浪费宝贵的人力资源和时间使用 PIC12 这么低级的产品；即使开发，代码也无人维护。

即使是比较简单的嵌入式应用，笔者认为参考一个操作系统来实现一个最小调度核心也是有益于开发的，最好具备或掌握以下几点。

- 分层的硬件中断，能够实现中断优先级。最好不要是 PIC16 那种共享中断源，ISR 还需要自行判断中断源的平台。
- 合理使用定时器中断，最好是参考 Cortex-M 实现 system ticker，这可以让子程序具备超时退出机制。
- 通信相关子程序，采用 8 的倍数实现环形缓冲区，配合中断服务实现主循环和通信堆栈之间的数据交换。
- 在缓存区竞争的情况下，需要使用临界区保护代码，关闭收发中断。
- 在最差情形下，主循环的循环时间应小于 128ms。
- 将各大任务模块根据有限状态机进行分割，放置在主循环中。
- 合理使用回调函数，实现数据在应用层和通信协议之间的传递。
- 如果能够实现简单的任务调度，则需要实现 wait/delay/sleep 中将控制权交给其他任务的算法。

最基础的嵌入式软件架构：main loop + ISR + callback + timer 可以实现较为复杂的程序设计，如 USB 驱动、TCP/IP 堆栈、AT MODEM 堆栈等。虽然 C 是一种比较贴近硬件的语言，但我们依然可以从抽象度更高的语言中学些面向对象编程，并利用结构体、函数指针等来实现。

目前，在笔者的嵌入式项目中依然有 70%的开发依赖于这种简单的开发模式。实际上，在各大半导体供应商提供的参考设计中，使用 RTOS 的也不占多数。不过随着物联网的发展、应用的日渐复杂，集成了更多中间件和多种通信堆栈，开发团队在市场和工程复杂度的双重压力下，日益依赖 OS/RTOS 以减少开发难度和复杂度。

4.10.2 RTOS 的优势

主循环+ISR 的配合，虽然可以完成大多数简单任务，但是有以下缺陷：

- 循环和 ISR 之间的数据交换采用全局变量完成，即设计者需要确保数据一致性；
- 复杂时钟设计较为困难；
- 必须要拆分超过主循环时间的耗时操作（比如 delay 密集型的函数）；
- 任务状态机拆分导致应用程序较难理解。

RTOS 将程序分成独立任务，每个任务单独都是一个循环，并按需调度：

- 任务调度——任务在需要时进行调用，从而确保了更好的程序流和事件响应；
- 多任务——任务调度会产生同时执行多个任务的效应；
- 确定性的行为——在定义的时间内处理事件和中断；
- 更短的 ISR——实现更加确定的中断行为；
- 任务间通信——管理多个任务之间的数据、内存和硬件资源共享；
- 定义的堆栈使用——每个任务分配一个定义的堆栈空间，从而实现可预测的内存使用；
- 系统管理——可以专注于应用程序开发而不是资源管理（内务处理）。

RTOS 的缺陷如下：需要额外的 RAM 资源做任务调度，并占用一些堆栈，同时需要 Debugger 支持 RTOS 任务级别调试。

RTOS 的供应商非常多，国内外 RTOS 的品种大约有 100 多种，比较出名的也有 20 多种。读者可以接触到的 RTOS 非常多： FreeRTOS/SafeRTOS，uC/OS-I/II/III，RT-Thread，ecos，QNX，RTEMS，VxWorks，Lynx，TI-RTOS，RTX，ThreadX，MQX……

挑选合适自己工程项目的 RTOS 还需要花费较大精力和学习成本，主要涉及：

- 版税和学习成本；
- 可靠性和整体设计合理性；
- 实时特性和资源占用；
- 开发调试工具链的支持；
- 中间件的支持、版税和适配成本。

此外还有一些操作系统不能够被称为 RTOS，其可以被归类为物联网、嵌入式相关的操作系统，如：

- 针对 WSN 的 Contiki 和 TinyOS；

- 针对 RFID/ICC 的智能卡操作系统。

在 RTOS 中，用户代码和操作系统大多是静态编译在一起的，往往也没有虚拟机的概念。接下来介绍几种常见的 RTOS。

4.10.3　uC/OS

uC/OS 是 Jean J. Labrosse 的操作系统，并成立 Micrium 公司进行 uC/OS 的商业化运作。uC/OS 并非开源或者免费许可证。开发者可以通过购买 Jean 的相关书籍或者合作芯片供应商开发板获得 uC/OS 源码。但如果要将 uC/OS 用于产品，则必须购买 uC/OS 商业许可证。

uC/OS 是国内可以获取的、发布时间最早的实时操作系统源码，具有较大的市场影响力。同时，uC/OS 也被适配到大多数 MCU/MPU 中，并拥有大量的配套中间件。

4.10.4　Keil RTX

作为 ARM 旗下子公司 Keil 的产品，Keil RTX 是免版税的确定性实时操作系统，适用于 ARM 和 Cortex-M 设备。使用该系统可以创建同时执行多个功能的程序，并有助于创建结构更好且维护更加轻松的应用程序。Keil RTX 实时操作系统内核组成如图 4-18 所示。

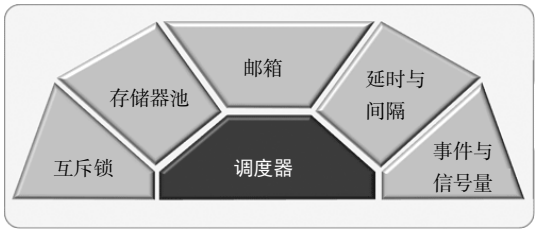


图 4-18　Keil RTX 实时操作系统内核组成图

Keil RTX 的几大亮点如下：

- 带有源代码的免版税、确定性 RTOS；
- 灵活的调度——循环、抢先和协作；
- 以低的中断延迟执行高速实时操作；
- 小空间占用适用于资源受限的系统；
- 不限数量的任务，每个任务都具有 254 个优先级；
- 不限数量的邮箱、信号、互斥函数和计时器；
- 支持多线程和线程安全运算；
- MDK-ARM 中的内核识别调试支持；
- 使用 μVision 配置向导的基于对话框的设置。

Keil RTX 可解决许多调度、维护和计时问题；同时，RTX 还支持以下中间件库：

- TCP/IP 堆栈；
- Flash 文件系统；
- CAN 驱动程序；
- USB 设备驱动程序；
- USB 主机驱动程序。

这些中间件采用商业版税，不免费。基于某些特定品种的 MCU 可以获得免费支持。

4.10.5 mbed RTOS 与 mbed OS

mbed 自带了 mbed RTOS 实时操作系统，实现了线程控制、线程同步、线程间通信、线程调度等功能。mbed RTOS 的实现建立在 CMSIS-RTOS 基础之上，CMSIS-RTOS 是 ARM 公司提供的用于线程控制、资源和时间管理的实时操作系统的标准化编程接口，可以方便各类实时操作系统的开发。

mbed OS 是面向物联网的操作系统，可以看到 mbed OS 不仅仅包含 RTOS，它还包括巨大的通信协议栈和各种中间件：IPv6/6LowPAN/FOTA/TLS/D-TLS/CoAP/ MQTT/HTTP，以及基础的事件框架和线程管理等。

此外，mbed 还包括配合合作伙伴提供云端服务及 yotta 构建工具。yotta 是基于 Python 的构建工具，专门用于 C/C++软件的自动构建。所以，mbed OS 可以包含 mbed RTOS 或者其他 CMSIS-RTOS 兼容的 RTOS，但本身提供更加完整的各类物联网堆栈设计。而 mbedRTOS 是 mbed OS 生态的一部分。mbed OS 系统组成如图 4-19 所示。

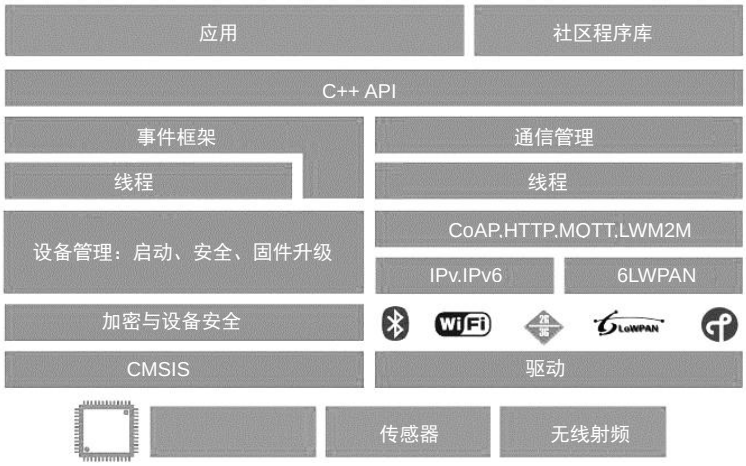


图 4-19　mbed OS 系统组成（物联网整体解决方案图）

4.10.6　FreeRTOS

FreeRTOS/OpenRTOS/SafeRTOS 核心是一致的。其中 FreeRTOS 和 OpenRTOS 的区别在于许可证，其分别对应 GPL/商业许可证。SafeRTOS 和 OpenRTOS 的区别在于：SafeRTOS 接受过严格的安全认证，并重新改写过。以上三种版本的共同特点如下：

- 免费的 RTOS 调度器，支持抢占式、协作式或者混合配置，并带有可选的时间切片调度。
- 衍生的 SafeRTOS 产品提供更高的代码完整度。
- 针对低功耗语言提供无时钟模式。
- RTOS 对象，包括任务、队列、信号量、软件定时器、互斥量和事件组，可以动态或静态分配 RAM。
- 占用的资源很少。
- 支持 30 多种嵌入式系统架构。
- FreeRTOS-MPU 支持 ARM Cortex-M3 存储器保护单元（MPU）。
- 小型化，简单易用。RTOS 内核二进制映像大约占用 4KB~9KB。
- 源码结构非常容易移植，主要使用 C 语言编写。
- 支持实时多任务和协作多任务。
- 为任务间、实时任务与中断同步的通信和同步，提供任务通知、队列、二进制信号量、计数信号量、递归信号量和互斥量。
- 独创的事件组或事件标志实现。
- 互斥量带优先级继承。
- 足够的软件时钟。
- 强大的执行跟踪功能。
- 堆栈溢出检测。
- 针对某些开发板，提供预配置 RTOS 应用，实现开箱即用的体验和快速学习曲线。
- 免费论坛支持，提供商业支持和许可证。
- 实时任务数量不受限制。
- 任务优先级数量不受限制。
- 对于任务优先级分配没有限制，可以为多个实时任务分配同一优先级。
- 免费多架构开发工具。
- 免费嵌入式软件代码。
- 免版税。
- 可以在标准 Windows 上进行交叉编译开发。

4.10.7 Linux 是开发复杂联网设备的现实选择

虽然我们可以找到大量的 RTOS 和中间件以实现一个完整的物联网设备，但是随着大量软件堆栈的引入，它越来越复杂，变得更像一台计算机，系统集成和调试也越发困难。

下面举个客户实例。最初用户采用的是 STM32F103+USB 主机，用于数字和模拟物理量数据采集。后来，陆续增加了串口 Wi-Fi、串口 BLE 模块、TF 卡存储、DNS 域名查询、TLS 证书以及时钟远程同步等。从工程师的角度来看，他们可能还想学习 mbed RTOS 之类的开发。但是从企业角度来看，系统越来越复杂，为何不直接采用计算机的操作系统呢？比如 Linux。除了定制的工业以太网对于实时性有要求，大部分 IP 通信并不是实时任务，这些可以采用 Linux 来完成，而将实时处理任务交给 MCU 来完成。大小核搭配，可以很好地兼顾 BOM 成本、复杂度和开发周期。

随着 ARM9/Cortex-A 系列处理器的价格快速下滑，Linux 系统越来越受到开发者的青睐。实际上，除了 Linux，开发者的选项并不多：Windows CE/Windows XP Embedded 退出了市场，VxWorks 等商业操作系统需要支付许可证费用。所以，Linux 开发是现实的选择。

基于 Linux 开发的确可以简化设计难度，但是 Linux 底层驱动开发对于开发人员的挑战却比一般 MCU 要大得多。

4.10.7.1 LFS/CLFS

LFS，即 Linux From Scratch，特指下载 Linux 源码，采用编译并安装 Linux 的方式。在 LFS 基础上，还衍生了多个版本。

- BLFS：Beyond Linux From Scratch，进一步完善 Linux 基本系统。
- CLFS：Cross Linux From Scratch，强化了交叉编译和嵌入式系统适配，包括 sysroot/embedded 两种方法。
- HLFS：Hardened Linux From Scracth，强化 Linux 安全使用。
- ALFS：Automated Linux From Scratch，编译和安装流程自动化方法。

LFS 的意义在于让使用者理解编译参数和软件补丁的作用，理解 LFS 附带的脚本的工作过程，从而达到独立制作并完善发行版的目的；而 CLFS 是构建嵌入式系统的主要方式。当半导体供应商设计一款 MPU 的时候，向客户提供的 BSP（Board Support Package）大多采用 CLFS 进行开发。

CLFS 构建的 Linux 的特点如下。

- 交叉编译内核：采用 GCC 交叉编译 Linux 源码。
- 交叉编译应用程序：除了操作系统，应用程序也需要进行交叉编译。
- 系统最小化：仅包括必要的 bootloader/rootfs/kernel/ramfs 和 busybox 等最少量组件。

CLFS 可以针对资源受限的 SBC 进行剪裁。许多嵌入式 MCU/MPU 可以外置 SRAM/DRAM，

但是缺乏 MMC，如 68K/ARM7TDMI，所以采用 uClibc 的 uCLinux 出现后，系统需要的最小 RAM/ROM 也不过 4MB 左右，实现了 Linux 在低端产品的普及。uCLinux 主要在嵌入式使用，ARM7TDMI/68K 退出市场后，可以在 FPGA、ADI Blackfin DSP 和 Cortex-M4F 上看到 uCLinux。除此之外，一些非常特殊的 x86 Linux 发行版也采用 uClibc/uCLinux 来构建以减少系统资源消耗。

现在，参考 CLFS 方法使用交叉编译器在 ARM MPU 上构建一个包括根文件和 Busybox 在内的嵌入式 Linux 最小系统，烧录在闪存、U 盘和 CD-ROM 中。其 RAM/ROM 大约在 64MB 左右。

由于资源很有限，最重要的 Linux 核心工具都静态编译在 Busybox 中。常见的工具都以符号链接的方式指向 busybox。默认状态下，LFS 以 C/C++为主要编程语言。其他高级语言（包括 Python）需要单独交叉编译才能够安装到系统中。这在许多情况下，并不是一件简单的事情。许多时候会遇到技术问题无人解答，令人抓狂。

即使在 CLFS 系统中，我们依然有许多办法运行 Python，可以使用 CPython 交叉编译、PyMite/MicroPython 交叉编译，以及 Java/Jython 的运行环境。总有一种办法可以运行 Python。配合 Linux 的设备文件，Python 应用程序可以很顺畅地运行于 CLFS 的 Linux 系统中。

4.10.7.2　嵌入式 Linux 发行版

使用交叉编译器对于开发者的要求门槛较高。读者如果阅读过 CLFS 建立交叉编译器的教程，就会明白交叉编译器的构建本身就是一个很烦琐的过程。如果对于一些软件进行过交叉编译，那么其中软件组件的依赖性也会让人头疼不已。构建整个系统既费时又费力，还容易出错。

CLFS 的主要目的是压缩资源占用，降低成本。随着存储器成本的下降，这些要素的优先级在降低。桌面和服务器领域的一些管理方法（比如标准化安装流程、各类软件管理工具等）被证明可以简化软件开发和管理的难度。所以，Linux 开源社区中开始出现了一些类似的解决方案。与 LFS/CLFS 相比，嵌入式 Linux 发行版依然可能采用交叉编译，但却由供应商采用预编译方式，并采用软件包管理工具进行安装。在这个领域，需要介绍以下几个开源项目。

1．Linaro

2010 年，Linaro 由 ARM 阵营的多家公司联合创建。作为非营利性工程组织，其负责为成员单位出品的 SoC 提供 Linux 支持。Linaro 的目标是为多个软件发行版提供稳定、优化、测试过的工具和代码，以减少嵌入式 Linux 软件的底层碎片化和持续优化。该项目所指的 Linux 软件包括核心操作系统、库、GUI、工具链以及应用程序。

Linaro 目前主要支持的架构为 ARMv7/ARMv8，并逐渐放弃针对旧架构的技术支持。

2．Yocto/OpenEmbedded

Yocto 是 Linux 基金会下属工作小组，其工作目标是为创建 Linux 发行版和独立于底层架构的嵌入式软件而提供工具和自动化流程设计。2011 年，Yocto 与 OpenEmbedded 共同维护工程，

并产生了 OpenEmbedded-Core。所以，Yocto 兼容 OpenEmbedded 项目。

Yocto 不是 Linux 发行版，而是一种开源协作软件。它提供了模板、工具和方法替开发者创建定制 Linux 发行版，并安装到嵌入式产品中，而无须过于关心硬件体系。Yocto 的构建工具是 poky/bitbake。其制作 Linux 发行版的流程如图 4-20 所示。

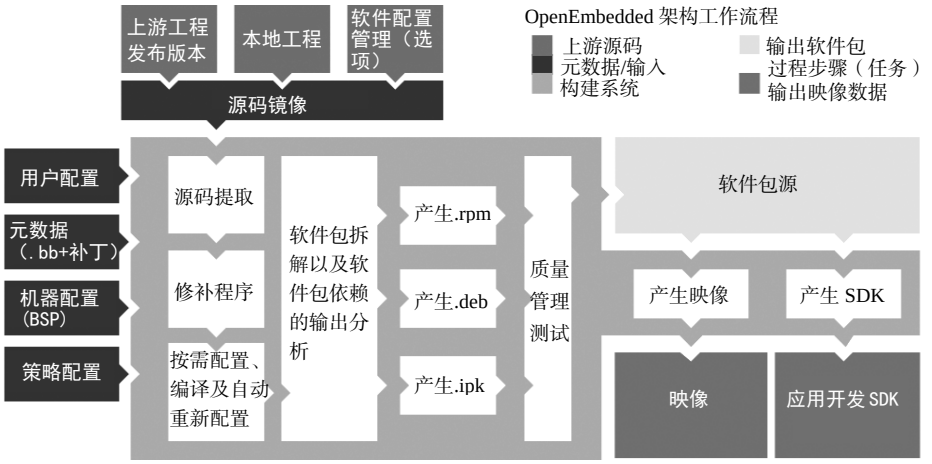


图 4-20　Yocto Linux 发行版制作流程图

OpenEmbedded bitbake 像所有的构建工具（比如 make、ant、jam）一样，是解决软件依赖性，并控制如何构建系统的工具。bitbake 收集和管理大量软件间依赖关系的描述文件（称为包的配方），然后自动按照正确的顺序进行构建。确切地说：OpenEmbedded 是一些用来交叉编译、安装和打包的 metadata（元数据）。

OpenEmbedded 已经被用来构建和管理很多的嵌入式发行版，包括 OpenZaurus、Angstrom、Familiar、SlugOS、OpenMoko。但 **OpenEmbedded 不是发行版**，OpenEmbedded 本身依赖于主机的 Python 进行构建。

在 GitHub 的 OpenEmbedded 工程中，有关于如何配置 Python 菜单及常见 Python 扩展库如 Twisted 的说明。读者需要按照某个平台，比如针对 Beagle 的 Angstrom 配置一次，这样可以充分体验一下 OpenEmbedded 的配置流程。

与 Linaro 仅限于成员单元的 ARM SoC 不同，Yocto 支持 ARM、MIPS、PowerPC 和 x86/x86-64 不同的硬件平台。

3．OpenWRT/Buildroot

OpenWRT 是基于 Linux 内核的嵌入式操作系统，主要用于路由器和网络设备相关领域。其主要的组件包括 Linux 内核、util-linux、uClibc 和 BusyBox。所有软件均针对运行所需的存储器做了优化，以适配到家用路由器中有限的闪存和 RAM 中。OpenWRT 的其他特性如下：

- 基于 overlayfs 的可写入 root 文件系统；
- 类似 dpkg 的 opkg 软件包管理器，软件库中包括 3500 种软件；
- LUCI 用于统一和简化系统配置管理（基于 Lua）；
- 网络通信相关软件包；
- USB 接口外接打印机、移动宽带 MODEM、网络摄像头、声卡等设备；
- 支持 SAMBA/NFS/FTP 文件传输、CUPS 打印服务、DLNA/UPnP A/V 数据流、Asterisk PBX 和 MQTT 协议。

OpenWRT 采用 Buildroot 构建工具来定制系统。该工具由 makefile 和补丁文件组成，可以实现 Linux 移植流程自动化。Buildroot 的特性如下：

- 易于在多个架构间进行移植；
- 使用 kconfig，即 Linux kernal config 做选项配置；
- 提供集成化编译工具，如 gcc、ld、uClibc 等；
- 提供抽象构建工具 autotools（automake、autoconf）、cmake 和 SCons；
- 自动化处理 OpenWRT 映像文件创建流程；
- 许多常见补丁。

Buildroot 的作者是 Peter Korsgaard，其最初的开发目的是为了 uClibc 的测试，后被广泛用作嵌入式 Linux 的构建工具。

基于以上开源组织提供的构建工具、软件管理器，可以大大简化嵌入式 Linux 构建流程。实际上，使用这些工具构建的嵌入式 Linux 发行版，大多支持 Erlang、Java、JavaScript、Lua、PHP、Perl、CPython、Ruby、Tcl 等各种编程语言。所以，Python 在此类预编译的嵌入式 Linux 中可以得到很完善的支持。

4.10.7.3　全功能 Linux 发行版

除了嵌入式 Linux 发行版，现在越来越多的 Linux 发行版采用了从标准 Linux 发行版进行剪裁的方式构建。这需要一个前提，即标准 Linux 必须支持多个架构。随着 ARM 架构越来越流行，处理能力越来越强，新的 Linux 发行版中除了 x86 平台，纷纷增加了 ARM/MIPS 处理器的二进制版本。可以剪裁成不同的配置，针对云计算、桌面系统、物联网、手机、平板等市场进行销售。它们的特点如下：

- 采用预编译软件库；
- 采用原生编译链编译软件；
- 采用软件包管理工具安装软件；
- 除了 busybox，还有大量独立软件。

在标准版的 Linux 发行版中，Debian 和 RedHat 是两个主要的 Linux 发行版，并各自衍生出

Ubuntu/Fedora/CentOS 等衍生发行版。各个嵌入式 Linux 发行版也都从这两支主线发展而来。比如 Pidora 来自 Fedora，Raspbian 来自 Debian 等。开发者可以使用 ipkg/opkg/deb、apt-get/dpkg、yum/rpm 等软件管理包来安装软件。

完整版 Linux 采用预编译的方式，即由 Linux 操作系统开发商完成编译，而开发者采用管理工具安装应用软件的模式。此类系统即使是针对嵌入式系统如 RaspberryPi，也都具备一个完整的操作系统发行版框架。和嵌入式 Linux 发行版采用交叉编译不同，完整版的 Linux 开始转向使用原生 GCC 进行编译。实际上 Raspberry Pi 就是在多台 ARM 主机编译池的原生编译器中产生的。

1．Ubuntu for ARM

Ubuntu 是目前使用最广泛的消费级 Linux 操作系统；同时 Ubuntu 还有服务器版，也非常容易定制。Ubuntu 14 还为中国用户定制了一个麒麟（Kylin）版本。相对于 x86 版本来说，其 ARM 版安装平台不算多。相信随着 ARM 平板电脑、手机、服务器领域的普及，Ubuntu for ARM 也将迅速普及。

相当多采用 Allwinner 处理器的嵌入式主板如 PCDuino 和各种山寨派等，也都将 Ubuntu ARM 版列入其支持的操作系统。Ubuntu/Debian for ARM 与桌面系统相比，其主要区别是使用了不同的架构。此外，在许多嵌入式系统中还需要一些特殊配置，以适配特殊硬件或减少不必要的系统消耗。

2．Ubuntu Core

Ubuntu 有意将 Linux 作为物联网的智能可扩展平台。Ubuntu Core 是面向智能设备的最新平台，并可以存储在本地或者运行云端相同的软件。从名字上就可以理解这个版本的用意，该版本是 Ubuntu 核心操作系统。

Ubuntu Core 更加轻量化，更加适合 ARM/x86 SBC 或者作为容器基础映像使用。Ubuntu Core 既可以运行在设备上，也可以运行在云端容器中。无论是 ARM 还是 x86 版本，Ubuntu Core 具备一样的 API 和安全更新。在 Ubuntu Core Snappy 文档中，列举了以下的运行平台。

- 设备：BBB、树莓派和 x86 设备。
- 本地：虚拟机实例。
- 云服务：Azure，Google，Amazon EC2，OVA image，Vagrant。

完整功能的 Linux 发行版中直接支持 Python/Java/JavaScript/Lua/Go 的编程。笔者认为这五种语言可以在物联网领域大放异彩。

CLFS/嵌入式 Linux 发行版/全功能 Linux 发行版操作系统的特性不同，Python 的实现程度也不同。但无论哪一种方式，都可以安装 Python 运行环境，加快应用程序开发。

4.11　物联网中间件

何为中间件？这个被用滥的术语主要用来描述一种独立的系统软件或服务程序。比如数据库、队列服务等。

在本章中，用于将除操作系统之外的重要的独立软件库归类于“中间件”。现在的嵌入式系统种类非常丰富，需要各种独立的中间件提供服务才能够构成系统。比如 TCP/IP 堆栈、WSN 传感器通信堆栈、文件系统、用户界面系统和终端等。

实际上，操作系统往往是这些中间件的基础。无论是 uC/OS、FreeRTOS，还是 RTX，都会提供一系列自己研发或者第三方的堆栈和软件包以简化用户开发的难度，并收取商业版税。商业版本的中间件版税成本较高，这也是免费开源的 Linux 在嵌入式系统中发展较快的原因，因为这些堆栈和软件包往往都是 Linux 自带的服务，可以实现标准化开发。以上提到的几类中间件产品，除了高度依赖硬件的 WSN 堆栈，大多数实现了标准化开发，开源设计也都非常多。

在 MCU 的产品线软件系统中，这些“中间件”都需要开发者仔细挑选评估，而且基于特定 RTOS 的选择项也不多，大多需要剪裁和定制。

4.11.1　WSN 堆栈

WSN 的选择太多了。由于存在太多的物理层、媒体访问层、链路层和通信调度策略与协议选择，因此几乎没有标准化的 WSN 中间件。此类堆栈大多封装在 RFIC/RF SoC 内部的 MCU ROM 中，比如 Nordic nRF51822 和其他的一些蓝牙低功耗 IC，都将 Cortex-M0 的低 64KB 作为堆栈 ROM，而高层用户采用剩下的 ROM，并使用函数调用来访问 ROM API。

该领域也有一些开源的 WSN 堆栈，如 6LowPAN、TinyOS、LoRaWAN 等，还有 9.3.3 节提到的 panStamp。读者可以自行参考源码，剪裁成自己需要的 WSN 堆栈。

4.11.2　TCP/IP

TCP/IP 作为互联网/物联网的标准，原本就是开源的。但是嵌入式 TCP/IP 需要不少优化，uIP 和 LwIP 是这方面的佼佼者。其作者是同一人：Adam Dunkel。LwIP 也已经被嵌入许多 Wi-Fi 芯片，成为主流选择。

此外，许多传统的 IDM 供应商如 Freescale 会配合 MQX RTOS 提供免费的 TCP/IP 堆栈。如果要将这些堆栈移植到其他平台，也需要花不少时间。

4.11.3　USB

USB 堆栈不是一个，而是若干组。因为 USB 需要分为主机、设备、OTG，还需要针对主

流的 MSD/HID/VCP/DFU 等不同的功能进行设计。并且由于 USB 往往需要依赖于特定硬件，因此目前跨平台的 USB 堆栈很少。即便是 ARM mbed 提供的 USB 堆栈，也仅仅支持 NXP/Freescale 和 STM32F4XX，更加常见的 STM32F103/L152/F072/L052 等需要使用 STM32 Cube 系列 SDK 来开发。

在这方面，还需要依赖于芯片供应商的技术支持。有能力的读者可以将底层驱动与高层的 Arduino/mbed C++ API 进行整合。

4.11.4 FAT/FS

FAT/FS 以及对应的 USB/SD 卡驱动在开源社群的多年努力下，也逐渐成熟，许多系统还开发出 MSD 的 Bootloader 以充分利用这方面的优势。这其中比较著名的开源设计有 FatFs。作者是日本人，居住于埼玉县上尾市。其项目主页中只有网名：ELM。最近登录他的网站，发现 FatFs 还在不断地更新，并支持 exFAT/FAT64。

4.11.5 GUI

在嵌入式的 GUI 中，QtEmbedded 比较有名气。此外，μGUI、μC-GUI 以及国内的 microWindows，也可以作为候选的中间件。针对更加低级的显示器件，如字符 LCD，或低于 QVGA 分辨率的 LCD，则往往需要定制 UI 组件。

4.11.6 Terminal

终端（Terminal）协议在嵌入式系统中还在发挥着很大作用。在终端协议中最著名的通信协议是 VT100。该协议简单有效，在 mbed 和许多源码中都已经存在开源代码，用户可以直接导入工程使用。

4.11.7 MQTT

MQTT 将在 9.4.5 节中进行重点介绍。十多年前 IBM 提出 MQTT 协议，到现在其成为物联网的事实标准协议。MQTT 得到了大多数编程语言和开发平台的支持。MCU C/C++语言固件及 Python 等高级语言，都支持 MQTT 客户端。不仅 mbed OS 官方支持 MQTT，许多开发者也针对 Arduino/mbed/RaspberryPi 提供了各种解决方案。

- https://developer.mbed.org/teams/mqtt/code/MQTT/；
- https://developer.mbed.org/teams/mqtt/code/HelloMQTT/；
- https://github.com/yilun/MQTT-client-on-mbed；
- https://developer.mbed.org/users/jwende/code/MQTT/；

- https://developer.mbed.org/users/Nim65s/code/niMQTT/。

传统物联网采用 TCP 长连接方式，MQTT 也同样采用 TCP 长连接方式。但是 MQTT 依托的是消息队列，可以在数据入库之前将数据转发给客户端和浏览器以及第三方应用，而不必流经数据库。其在实时性上要远胜传统方式。

4.11.8　CoAP

由于物联网中的很多设备都是资源受限型系统，只有少量的内存空间和有限的计算能力，所以传统的 HTTP 协议应用在物联网上就显得过于庞大而不适用。IETF 的 CoRE 工作组提出了一种参考 REST 架构而设计的 CoAP 协议。CoAP 采用了 UDP 做传输层，适合资源非常有限的嵌入式系统。

CoAP 在应用上与传统物联网应用有所不同。CoAP 往往在边缘设备中实施“服务器”，而云计算集群是作为“客户端”存在的。在数据变化时，边缘设备服务器才推送数据到云计算集群。这种应用模式在直连拓扑中很实用，降低了系统功耗，减少了数据流量。如果服务器和客户端在同一内网中，则还可以利用组播方式进行 1∶N 的数据分享。但如果边缘设备在企业内网中，数据库基于公网云服务器，则需要端口转发等技术来解决内网穿透问题。

另外一种就是配合 Web 服务器，将 CoAP 作为代理服务器的应用方式。边缘设备作为客户端，CoAP 做代理服务器，Web 作为后端服务器连接数据库。代理服务器起的作用是 CoAP/HTTP 的协议转换。其最大的技术问题依然是需要克服内网穿透。

所谓 CoAP 的 REST 架构，有两种含义：

- 在设备端服务器中，可以把物理量采集点视为“资源”进行读/写；
- 在代理服务器中，可以把云服务器视为远程数据库“资源”进行读/写。

CoAP 是 UDP 版的 REST，将 CoAP PDU 承载于 UDP 套接字上即可以实现收发数据，技术实现上的确要容易许多。但是 mbed/GitHub 网站上的 CoAP 实现相对要少得多。总结下来有以下几点原因：

- CoAP 使用的 UDP 虽然轻省，却不利于数据和控制流的双向传输，需要在 CoAP 数据报基础上维持连接。
- CoAP 和 MQTT 虽然都采用了代理服务器的模式，但是 MQTT 本质上是消息队列，而 CoAP 是纯粹的代理服务器，没有解决公网的数据分享瓶颈。
- CoAP 对于云计算服务商的重要性没有 MQTT 重要，基于普通 IaaS 即可以实现。
- CoAP 在设备端实施服务器的做法，多数开发者并不熟悉。
- 基于 TCP 长连接的 CoAP 尚在草案中。
- CoAP 更加适合 WSN 内部通信。

在目前的物联网应用中，MQTT 日渐成为主流选择，但 CoAP 却是被低估的一种方式。

4.12 物联网安全性

对于物联网传感层的安全性，笔者之前体会得不算很深。但是笔者在看过某厂家描述的安全漏洞之后发现，物联网安全是一个必须要高度重视的环节。哪怕是一个简单的只读性质的温度传感器依然如此。假设这个温度传感器遭到破坏，黑客通过传感器网络侵入系统，伪造了一个传感器异常的过热信息给中心服务器，那么服务器很可能会触发喷洒系统操作，引来仓库内或者住宅内“水漫金山”，造成实际的经济损失。

在国内著名的乌云网中，解释了如何侵入现在常见的 BLE 系统，比如手环、成人用品、支付系统等，这也都会对这些系统造成直接的（如固件损坏，资金被盗取）、间接的（如隐私、个人信息流失）、衍生的（随后发生的附加损失，比如引发的争吵、银行账户失窃、手机和计算机被加密劫持）各类损失。

物联网安全正在成为最弱以及亟待加强的一环。现阶段，黑客生态链目的明确，针对性强：以获取个人信息和资金为主要目的；而国内外各种敌对势力则以破坏金融、基础建设和工业设备为目的进行攻击。所以，利用各种方式维护设备、通道和服务器安全是非常重要的。

4.12.1 安全相关芯片

物联网，尤其是智能硬件风潮之后，行业内比较关心的是联网安全。互联网发展到今日，受到电子商务和金融行业的市场驱动，已经存在一整套安全算法保证交易的顺利进行，SSH/SSL/TLS 已经成为工业标准。对称/非对称算法等也都得到大量使用。

作为行业内的标杆，基于大素数分解的 RSA 算法从 1977 年迄今经受了三十多年的黑客攻击。计算机发展使得这些算法在 GPU 计算、云计算面前变得越发脆弱。银行业使用的 RSA 算法以 2048 位为基线，而 4096 位也已开始使用。与 2048 位 RSA 处于同等数量级的算法还有 224 位 ECDSA、SHA-256 等。

这些算法在一些物联网的局部应用场景中还都是足够强壮的。但是，这些算法在资源受限的设备端的实施则受到了各种限制，主要是成本和芯片面积的限制。所以，需要外部芯片的配合。

4.12.1.1 IC 卡芯片

物联网最初的应用是 IC Card/RFID 应用领域。在这方面，许多半导体公司都已经推出了成熟产品，基本上是以下几类：

- ISO7816 智能卡，如银行使用的接触型智能信用卡、手机中的 SIM/UIM/USIM 卡和 eSIM 模块及机顶盒 CA 卡；
- ISO14443 非接触型智能卡，如 NXP 的 Mifare RFID 和 Mifare Pro 双界面卡；

- USB 接口安全模块，用于网银令牌 U 盾等；
- POS 机内的 SAM 模块（采用 ISO7816 或者其他接口封装）。

这些安全产品往往内置加密协处理器，可以支持 AES/3DES/RSA 算法，例如 NXP 的 IC Card 产品线的加密协处理器已经从 512 位推升到 4096 位。内嵌卡片操作系统（Card OS）通常采用掩膜工艺固化。之所以在芯片中内置加密协处理器，基于以下两点原因：

- 安全需求。内置于芯片内（甚至是芯片的不同布局夹层内）以减少外部侵入计算的可能。
- 计算能力不足。通用架构处理器处理 RSA 算法颇为吃力，必须使用专门的硬件做 RSA 加速。

4.12.1.2　独立安全芯片

此外，为了迎合物联网的需求，一些半导体厂家如 Atmel 和 Dallas/Maxim 提供独立的网络安全相关硬件产品。

Atmel 提供的安全产品品牌为 Crypto，包括：

- 保密认证（Crypto Authentication），支持安全哈希算法（SHA-2）、高级加密算法（AES）、椭圆曲线加密（ECC）算法；
- 可信平台模块（Trusted Platform Module），支持 2048 位 RSA 算法；
- 加密 RFID（CryptoRF），ISO14443 接口存储型 RFID；
- 加密存储器（CryptoMemory），EEPROM 或者 ISO7816 接口。

Maxim 提供的嵌入式安全产品品牌为 DeepCover，包括：

- 安全管理器（Security Manager），保存安全敏感信息；
- 安全认证器（Security Authenticator），用于保护电路设计知识产权；
- 安全微控制器（Security Microcontroller），用于安全算法和控制，可以支持 3072 位 RSA 算法。

基于安全理由，市场上很难寻找到独立封装的 RSA 协处理器 IC。即使有，也需要签署 NDA。我们可以利用一些基于 FPGA 的 Verilog/VHDL 开源工程。虽然这些 IP 可能性能有限，比如仅支持 512 位 RSA 而非 2014 位 RSA，但却可以作为一个研究起点。

4.12.2　安全中间件

安全领域也有开源软件项目：AVRCrypto/ARMCrypto。这两个项目是类似的，分别针对 AVR/ARM 进行了优化。开发者为同一团队。项目中收集了多个加解密算法和哈希算法；不过受限于硬件，其支持的位数相对有限。

ARM mbed 提供标准的 SSL/TLS 产品，其前身是 Polar SSL/TLS，它是一种面向资源受限

计算平台的商用闭源 TLS 软件包。ARM 收购 Polar 公司后将其开源。无论是采用 MQTT/CoAP/REST API 还是私有的 TCP/UDP 协议，基于 ARM mbed 的物联网设计均可以使用标准化的传输层安全技术。

此外，一些小型供应商如 Oryx Embedded 公司提供各种中间件，其品牌为 Cyclone（请不要与 9.7.10 节中介绍的 Cyclone Web 框架相混淆），包括 CycloneTCP、CycloneSSL 和 CycloneCrypto。虽然该公司提供商业服务和许可证，但是其开源版本也可以使用。

4.12.3 Python 安全算法

Python 是一个非常棒的验证算法平台，针对加解密算法有一些专门的包。纯粹的 Python 算法效率不高，好在可以使用相对应的 C 扩展包。不过需要注意的是，不同的加解密算法包可能存在一定的兼容性问题。

4.13 设备固件更新

现在许多电子设备都会进行固件更新。小到手机应用 APP，以及 Android 平台固件，大到 Tesla 汽车平台固件，甚至月球探测设备的固件都会通过远程方式进行更新。而这之前，大多数设备都需要回收到供应商处进行固件升级，或者依赖用户进行 DIY 更新。固件更新被归入产品生命周期管理范畴中。

4.13.1 固件更新技术发展史

在半导体的发展过程中，程序固化流程也经历了一个缓慢进化的过程。

最初的半导体芯片 ROM 芯片采用的是 Mask 掩膜工艺。这决定了程序的最小改动成本就是一整块晶圆，这至少需要 2000 美元。所以向半导体工厂下掩膜订单要非常谨慎，只有固件成熟了才会掩膜，这里开发者的风险非常高。

同时期在实验室设计阶段采用的是 UV-EPROM，即紫外线可擦除可编程只读存储器。这种存储器可以利用紫外线通过芯片的窗口擦除并重新编程。但是，EPROM 的芯片封装必须是陶瓷封装和石英玻璃窗口。所以小批量成本比较贵，仅用于样机。

塑料封装的 EPROM 只可以编程一次，即 OTP 存储器技术。如果采用稀释的酸将芯片塑料封装溶解，则可以利用紫外线擦除 ROM 内容后再编程。由于塑料封装成本较低，因此许多小批量订单均采用 OTP 技术交付。这时候，开发者用 UV-EPROM 做开发，用 OTP 投入市场。

EEPROM（电可擦除可编程只读存储器）的出现大大降低了嵌入式软件开发成本，但是这时 EEPROM/NOR Flash 的价格比较贵，直到 Atmel 将 Flash 80C51 的价格降低后其才开始普及。

在 CD-ROM 刻录机标准竞争最激烈的时代，许多半导体厂家为了应对快速迭代的 CD-R/CD+R 技术，纷纷推出了带 ISP 功能的 Flash MCU。采用 NOR Flash ROM+ISP Bootloader 成为嵌入式 MCU 的标配。

固件更新所需的工艺与技术迭代由此经历了整整 50 年。半导体、物联网跟互联网相比之所以迭代速度慢，其工艺的限制是一个很大的因素。

即使是现在，一些供应商出于安全、功耗和成本考虑，还在采用比较旧的工艺。IC Card 采用 Mask 工艺是出于安全的考虑。某些 MCU 采用 Flash 工艺，但是没有 Bootloader，这提供了系统设计灵活性，但是却减少了用户程序的存储空间。也有出于功耗考虑的设计，比如小米手环采用的 Dialog Semiconductor，其提供的蓝牙芯片 DA14580 的蓝牙堆栈就是采用 OTP 做用户程序空间。

所有这些技术细节，必须先联系原厂做出综合考虑。

在物联网时代，快速迭代是所谓精益开发的基本要素之一，固件更新是在系统设计之初就必须考虑的。而固件更新需要根据应用场景进行规划和设计。

固件更新需要考虑的几个重点：

- 采用本地固件更新还是远程固件更新？是线，还是无线？
- 固件缓存在物联网的哪一个环节？是服务器、网关，还是节点本身？
- 固件的版本识别、认证、授权、传输安全策略与传输后校验？
- 固件更新是否需要增加额外的存储器？
- 固件更新后如何进行个性化处理及参数设定？
- 更新固件的设备与层次？是节点，还是网关？Bootcode，一级 Bootloader，二级 Bootloader，固件更新如何利用硬件平台的存储、应用内再编程 IAP 能力？
- 固件更新如何利用硬件平台能力？
- 固件更新的加解密算法和授权？
- 固件失败的对策？固件回滚方式？

所有这些策略需要设备端和服务器端都做相应的修改。根据笔者的从业经验，可以针对此环节提出一些建议：

- MCU 采用内置 Bootloader 的品种，因为这是系统错误锁定后的唯一挽救方式；
- 结合应用设计 ISP/IAP 进行固件升级；
- 小型 MCU 固件，ISP 无须辅助存储器直接升级，ISP 程序必须定制；
- 大型 MPU 固件，固件可以先缓存在外部闪存中，校验后进行更新；
- 慢速或无线传输的固件，推荐采用保存外部存储器的方式以减少错误和重试次数；
- 固件需要版本识别、授权、校验，服务器记录归档；
- ISP 升级失败后，可以再次进入 ISP 流程；

- 最后的解决手段是退回制造商处，采用原生 Bootloader 和 JTAG/SWD 更新。

4.13.2 本地固件更新

本地固件更新主要用于生产环节、维修环节，以及有一定动手能力的用户自行升级。本地固件和 MCU/MPU/SoC 的连接能力密切相关，主要升级手段如下：

- 通信串口（ISP 下载）；
- USB Device（USB DFU/ACM/CDC/HID/MSD）；
- USB Host（USB MSD U 盘读取）；
- SPI/SD/TF 卡；
- 以太网（本地网络升级）；
- CAN 总线升级；
- I2C 升级；
- JTAG/SWD 固件升级；
- 终极手段：原厂 Bootloader 升级。

4.13.3 远程固件更新

中国台湾省台北市的 YouBike 工程（又称 UBike），2016 年 8 月底遇到了大规模升级的难题。运维团队拿着笔记本电脑在整个台北市逐一升级自行车固件。这个真实案例可以让读者了解远程固件升级缺失的代价。物联网设备一旦更新失败就可能必须花费巨资回收或派员实地一一修理，代价高昂。具体报道可以参见本章延伸阅读部分。

远程固件可以通过 IP 网络或工业现场总线进行升级。可以利用以太网或 RS485 等固网连接，但是无线方式正在成为主流。传感器网络固件空口升级（Firmware Over The Air，FOTA）和基于 IP 网络的 OTA，将是未来主要的固件升级模式。不管是何种网络通信方式，远程固件更新因为网络的参与而带来了更多需要考虑的问题：

- 更新速率慢；
- 失败概率增加；
- 容易被攻击；
- 固件更新算法本身也可能需要更新。

4.13.4 固件升级定制

在基于 MPU/Linux 的系统中，如手机、平板等，固件升级往往采用从远程下载固件，保存在本地存储器中，校验核对后进行升级。其整个过程相对简单。比较复杂的是 MCU 系统中的

固件升级。MCU 系统中一般有 ISP（In System Programming）和 IAP（In Application Programming）两种升级方法。两种方式的区别在于：

- ISP，无须专用编程器即可以下载固件，主要指的是通过本地 UART/SPI/I2C 等有线连接下载方式，大多作为 MCU Bootloader 存在；
- IAP，用户可以对固件进行编程，主要指用于烧录闪存的功能，有别于业务逻辑相关的用户程序。

启动顺序如下：

```
ISP Bootloader →  IAP → User Firmware
```

开发者需要针对不同的应用环境要素，如传输协议、IP 地址、端口等定制 IAP。如果在 IAP 中实施完整的通信堆栈（USB/TCP/IP/WSN），这与用户程序中现有的通信堆栈存在功能冗余，非常耗费 ROM 空间，解决的方式是设计可以复用堆栈的 ROM API。另外一种方式，就是利用外部存储器缓存，然后重启进入 IAP 后，由 IAP 负责闪存更新算法。总之，以硬件冗余换取代码的简化。笔者在一些设计中就通过 I2C 总线外接 FeRAM 来实现固件升级。固件升级后，FeRAM 可以作为用户数据存储器使用。此外，1MB 左右的 SPI 闪存价格便宜，也可以用于固件保存和升级。

出于安全考虑，一定的硬件冗余是必要的，增加不了多少成本。**工程维修人员的往返差旅费或快递费远远超过这些硬件成本！**

4.14　各类串口实现联网

在“万众创业，万物互联”的口号下，许多企业都非常急于将现有设备和产品升级。许多设备并没有通信手段，而添加通信 SoC 可能是一件耗时的工程。所以市面上大量出现了所谓串口“透传”模块，帮助设备实现蓝牙、Wi-Fi 连接。

4.14.1　串口协议的选择

利用 GPIO 和各类串口是升级现有设备进行联网的有效途径。其主要的物理层协议为 UART/SPI/I2C。这三种协议都可以使用 2~4 组 GPIO 进行模拟。

- UART：TX/RX，多见于 MCU 之间的通信，大多数 MCU 内置 UART/USART，速率至少为 115.2kbps，高速率品种可以支持到 1Mbps，软件模拟 UART 也可以达到 9600bps。
- SPI：CS/MISO/MOSI/CLK，这和 CLK 时钟频率有关，多见于 MCU 与高速外设间通信。除了 CS，其他三个引脚可以与其他应用复用。
- I2C：SCL/SDA，这和 SCL 时钟频率有关，可以实现多台设备并联，可用于 MCU 之

间或总线速率 400kHz 以下的外设通信，SCL/SDA 都可以复用。

- USB：多数为 USB Device，少部分为 USB OTG，可以在主机和设备间切换。

其中，UART/RS232 是大多数智能 MODEM 的基础，需要重点关注。所谓智能 MODEM 指的是内置 MCU，支持 AT 指令集或者类似指令集的 MODEM。

在一些工程实践中，笔者发现不能够过于简单地把串口协议承载到 TCP 数据通道上。虽然串口通信也是数据流通信，但却是基于物理层的通信协议，所以，短距离点对点传输的收发延时很小。而 TCP 是一种基于网络层的逻辑通道，由于中间节点缓冲转发的缘故，因此会出现粘包和半包（也称断包）的现象。此外，偶尔会出现多包和少包的现象。

这对接收端（往往是服务器端）来说有时判断存在困难。所以，必须规划一个合适的帧结构以切割粘包现象。除了要有报文头、报文结束，还应该有报文长度信息，这也是很重要的参数。有必要的话，还需要加入转义符和填充字符等。笔者将在第 9 章中介绍相关内容和参考解决方案。

4.14.2 模拟串口设备

在开发实践中，经常发生软件开发时硬件没有就绪的情形。我们可以利用 pyserial 和 USB 串口转接板来模拟串口设备。

以下就是笔者用来模拟 ZTE MC8360V2 的 AT 指令集的代码。

```
import serial
import datetime

def response(ser, res):
    ser.write(res)
    print(res)

debug_info = """
1x Engineering
State:Active
SO:0
Channel:201
Band Class:0
SID:13844
NID:2
Base ID:8772
PN:38
P_rev:6
MCC:646
MNC:66
Latitude:0.0902
Longtitude:0.4558
```

```
Rx Pwr:-61dBm
Rx Ec/Io:-7.0dB
Rx FER:0.0%
Tx pwr:-9.2dBm
Active Set:
1,38 -5.0
Neighbor Set:
19,256 -31.5,106 -31.5,100 -31.5,442 -31.5,268
-31.5,424 -31.5,18 -31.5,78 -31.5,434 -31.5,98
-31.5,88 -31.5,242 -31.5,306 -31.5,146 -31.5,274
-31.5,208 -31.5,506 -31.5,176 -31.5,238 -31.5

OK

"""
ser = serial.Serial('COM52', 9600, timeout=1)
while (ser.readable()):
    line = ser.readline()
    if line:
        print line

    if len(line)==4 and "AT" in line:
        response(ser,"\r\nOK\r\n")
    elif "ATE0" in line:
        response(ser,"\r\nOK\r\n")
    elif "AT+CPIN \r\n" in line:
        response(ser,"\r\n+CPIN: READY\r\n")
    elif "AT+CSQ" in line:
        response(ser,"\r\n+CSQ: 25\r\n")
    elif "AT+CNMI=2,2" in line:
        response(ser,"\r\nOK\r\n")
    elif "AT+CMGF=1" in line:
        response(ser,"\r\nOK\r\n")
    elif "AT+ZVCF=" in line:
        response(ser,"\r\nOK\r\n")
    elif "AT^PREFMODE=" in line:
        response(ser,"\r\nOK\r\n")
    elif "AT^VOLT" in line:
        response(ser,"\r\n^VOLT: 3705\r\nOK\r\n")
    elif "AT+1XDEBUG" in line:
        response(ser,debug_info)
    elif "AT+CCLK " in line:
        current = datetime.datetime.now().strftime('%Y/%m/%d,%H:%M:%S')
        current = current[2:]
        response(ser,"\r\n+CCLK: \"%s\""%(current))
```

通过 RS232/UART/USB 连接主机，在主机上运行该脚本，主机可以模拟对应的蜂窝数据

MODEM，加速嵌入式软件的开发。事实证明：用 Python 开发串口比用 C/C++开发串口简单得多！模拟 MODEM AT 指令集很容易，在 DTE 中使用 Python 开发固件难度很低，迭代速度也更快。

4.14.3 其他类型虚拟设备

利用 Python 不仅仅可以访问串口，还可以访问 SPI 和 I2C 总线。无论是通过 UART 桥接，还是利用 Linux 上的 Python 或者是 MicroPython 都可以实现这两种总线的模拟。这些是利用硬件来模拟设备。

除了硬件模拟，SocketCAN 的 Python 包还支持虚拟设备设计。因为并不是所有 SoC/MCU 都具备 CAN 总线，所以 SocketCAN 在测试套件中增加了一个虚拟 CAN 卡的设备，让开发者可以在没有 CAN 硬件的情况下开发应用程序。这个软件包值得仔细研究一下。

除了硬件和软件进行模拟，网络设备的模拟就更加简单了。用 Python socket 制作一个客户端配置客户协议就是一台虚拟设备。这可以构成一个闭环的开发链条。

4.14.4 ISP 编程器

许多功能复杂的 MCU/MPU，不仅芯片自带的 Bootloader 有多种模式，还可以另外定义二级 Bootloader，例如 uboot 就是常见的二级 Bootloader。但在芯片启动时，最初的 Bootloader 通常采用 UART 进行固件下载。早期的某些 MCU 甚至没有调试口，就靠用户用串口打印进行调试和下载固件。

使用 Python 作为 ISP 上位机，通过串口将固件下载到芯片中，是常见应用。Freescale Kinetis 和 NXP/STM 不同，属于没有 Bootloader 的“裸机”，需要客户自己通过 SWD 下载，或者自行设计 Bootloader 进行 ISP 下载。笔者曾经为 KL25Z 提供了完整的串口 Bootloader——USB Device Bootloader。其中，串口 Bootloader 提供了对应的 Python 上位机程序，并在 Freescale 的社群提供给开发者使用。

利用串口 Bootloader，大多数情况下都可以做到不开设备外壳就可以升级固件。这一点，NXP 的 LPC 系列 MCU 已经做到了完美。其内部 RST/ISP 引脚可以直接对接串口的 RTS/DTR，电平极性是兼容的。ST 的 ISP 引脚（BOOT0/1）采用了反转的电平，可以利用三极管或者逻辑门来反转电平。这虽然增加了一些成本，但其依然不失为一种通用的升级方式。

注意 带 USB Device 接口的 MCU 可能支持 MSD/HID/DFU 模式下载。需要仔细研判 MCU 的 Bootloader 构成。Freescale KL25Z 没有自带 Bootloader，所以客户可以采用串口 Bootloader、MSD Bootloader、HID Bootloader 以及 DFU Bootloader 中的一种。部分 STM32 USB MCU 并不支持 DFU Bootloader，只有串口 Bootloader，所以如果需要 DFU，则还需要自己设计 DFU Bootloader。NXP 的 LPC 基本上以串口 Bootloader

为主。

以上的 Bootloader 往往还需要使用对应的客户端，可以使用 Python 开发，也可以使用第三方的工具（例如 OpenMoko 的 DFU 工具）。其他物联网常见的 Bootloader/ISP/IAP 还包括 USB MSD 主机、以太网等方式。这方面需要和半导体供应商保持紧密联络，以获得足够的技术支持。

4.14.5　串口设备监控器

4.3.5.7 节提到了 LPC8XX 系列，笔者曾经围绕 LPC8XX 做过一些小型仪器。例如，在 M2M 项目中，笔者就用 LPC8XX 来做串口监视器。在典型的 M2M 应用中，MCU 和 M2M 模块间采用 AT 指令集进行通信。如何监控它们之间的通信是一件让人挠头的事情。用示波器看，只能看到波形；用逻辑分析仪看，需要某一路跳变事件作为触发信号，而且观察起来也不直观。所以，笔者使用了 LPC8XX 的三路串口进行监控：

- UART1 的 RXD 连接 M2M 的 TXD，波特率为 9600bps 或 115200bps；
- UART2 的 RXD 连接 MCU 的 TXD，波特率为 9600bps 或 115200bps；
- UART3 的 TXD 连接 PC 监控端的 RXD，波特率为 115200bps 或更高。

通过这种配置，无论是 M2M 还是 MCU 发送的数据，开发者都可以在 PC 监控端看到对应的字节，由 Python 脚本在本地生成时间戳，并在桌面分析程序中按照时间先后关系进行呈现。UART3 还有一种作用，就是对设备进行动态配置，使得 LPC8XX 在任意引脚上都可以监控目标串口，同时对波特率等参数进行配置。这种设计比逻辑分析仪实时性更强，更加直观。

4.15　本章小结

物联网与设备紧密相连，所以本章中罗列了嵌入式系统中常见的硬件平台、软件编程、操作系统、中间件以及开发工具等，希望能够带给读者更加全面的行业知识和开发手段的全局视角。同时，在物联网中有必要对系统互联技术有所了解。在第 5 章中，会针对物联网设备端中读者最关心的连接技术，包括组网和联网技术进行多层次的分析和讲解。

第 5 章

设备连接和编程接口

5.1 设备连接概述

第 4 章介绍了设备端开发环节中的嵌入式系统开发基础知识。本章则单独围绕嵌入式系统连接能力（connectivity），包括设备互联、设备组网和设备联网的各种实现方案进行介绍，并就如何使用 Python 进行编程具体展开介绍。

不同平台操作硬件的方式存在较大差异，部分代码可以兼容 Linux/Windows。但除非特别说明，本章主要以 Linux 平台，包括嵌入式和完整版的 Linux 发行版为主要开发平台进行介绍。基于 MCU 的嵌入式 Python 虚拟机因为与 Linux 存在较大差异，将单独在嵌入式 Python 虚拟机中进行介绍。

5.1.1 嵌入式系统连接层次

前面介绍了物联网发展的第三阶段是智能互联阶段，需要解决设备联网问题。这是目前大多数制造业企业的重点关注方向，具体有三大类问题。

- 设备连接：即智能化升级，是否需要为现有设备增加传感器？通过何种方式添加何种传感器？
- 设备组网：设备间是否需要组网？采用何种方式组网？
- 设备联网：如何将设备通过合适的接口（通信堆栈、通信协议和编程接口）进行联网？
- 这三个问题是系统连接性的三个层次，分别针对设备内和设备间的直接连接、设备节点间，以及设备与服务器之间远程联网的通信层次。如果再展望到物联网的第四阶段产品系统、第五阶段产品生态体系，还会出现更高层级的互联互通问题。比如：服务器间的数据共享，用户端与设备端互联，设备近端组网，大数据共享，等等。

不过就设备端的嵌入式开发，我们在本章仅仅关注以上三个层面的技术问题，并利用 Python

来解决这些技术问题。

5.1.2　选择正确的连接方案

有句话说得很好：“不要在错误的方向上飞奔。”

笔者在与诸多企业对接项目的时候，发现传统制造业在选择连接方案时存在严重的选择障碍。这些企业虽然在自己的专业领域具备相当的技术实力，但是缺乏对于各层级连接方案的足够了解，所以会在工程启动初期向许多供应商进行咨询。然而反馈越多，越难决定。

笔者经手的工程中有个很典型的例子，工程甲方分别向手机模块供应商、电信运营商、工业系统集成商、半导体供应商和设计公司，以及物联网智能家居系统集成商进行技术咨询和报价。由于这些供应商提供的技术方案南辕北辙，报价和服务方式也是千差万别，导致工程甲方犹豫不决，以致工程无法启动。实际上，这些供应商的行业背景完全不同，因此他们能够提供的服务层次、给出的答案必然千差万别。而工程甲方对于自身定位不清晰，这些项目最后起决定作用的往往是商务和人际关系，而将技术、成本和市场因素放到了次要的地位，这样很容易选择错误的方向。

对于那些自身定位不清晰，总是希望成为平台的传统制造企业，笔者觉得根据自己的产品定位落实物联网的连接技术方案是第一步，在工程初级阶段就规划“平台”是不切实际的。需要分成几个步骤完成：

（1）清晰地了解并定义需求，包括设备的地理位置分布、系统功耗、连接可靠性，以及数据特征（包括流向、流量、频率、延时等）、成本、实施周期等几个约束条件，再来对比供应商提供的方案做出折中选择。这是对于方案及供应商的选择原则。

（2）了解系统连接的三个层次：设备连接、设备组网、设备联网。如果是离散性设备（即地理位置分布得比较离散，比如电梯、挖掘机、车辆），可以采用设备直接联网。这个选择对于开发周期和运维成本影响较大。

（3）所有的系统连接性最终都具体落实到底层设备及芯片连接。采用 C/C++/Python/Lua/Java，最终通过 SPI/UART/USB 来实现接入系统。

（4）系统连接性的三个层次间存在一定技术依赖性。采用行业通用的标准化设计可以减少企业的投入和重造轮子的麻烦。

5.1.3　具体落实连接设计

企业明确定义并制定了以上各方面战略之后，选择合适的接口将方案落实到硬件、固件和软件设计中就变得容易了。

接口（interface）这一词汇在各个行业非常容易混淆。一台设备要联网，要考虑并挑选以下

几种接口。

- 物理接口：设备从控和主控芯片可以支持的物理接口如 UART/SPI/USB，这方面的可选项已经趋向集中。
- 通信接口：设备与外部服务器或者网关间通信的接口，特指通信协议如 6LowPAN/MQTT，包括有线连接和无线连接的空口协议。
- 编程接口：编程语言控制主芯片和通信的编程接口，包括操作系统、协议栈、中间件和语言调用接口。这部分可能是读者最关心的部分。

这三者间，存在着许多技术依赖性和技术细节。这也是传说中的“坑”。有些“坑”，一旦踩到，就必须要找到高手来铺平道路。好在现在国内发达的电子行业配套市场和方兴未艾的创客开源运动，使得许多现成的模块可以直接用于参考设计、产品开发甚至工程实践中。

5.1.4 本章内容安排

回溯五十多年的发展史，计算机系统出现了各种各样的物理端口和通信协议。随着企业间的优胜劣汰，以及持续不断的系统整合，总的技术趋势是高速化、标准化、串行化。

本章按照规模从小到大、从底层到高层的原则，罗列了多种连接技术，并单独将无线、蜂窝数据移动、工业应用进行分类介绍。然后，针对各类计算机系统中曾经流行与现有的行业标准中常见的物理端口，通信协议、编程接口，以及对应的 Python 软件包相关编程进行介绍。

为什么笔者还要介绍曾经流行的接口呢？因为技术演进是有传承的。一种古老的连接技术很有可能在某些特定环境中又可以焕发出第二春，创造出意想不到的应用：比如基于音频线路的 MODEM 接口，其一度成为手机支付企业（如拉卡拉）的主要连接手段，堪称是对于 NFC（Near Field Communication，近场通信）的逆袭。此接口还被用于照片快门等应用。

5.2 连接能力汇总

微电子/半导体行业为各行各业提供了异常丰富而多样化的产品线，体现了物联网连接能力（connectivity）的多样性。随着物联网、移动通信和智能硬件的发展，MEMS 传感器等新类别元件的演进非常快，变化也非常多。为了满足工业、农业、医疗保健、科学研究、国防需求，以及快速发展的移动通信和消费电子行业需求，半导体行业为这些行业提供了大量 MCU，以及各类定制的 ASIC/SoC 和模块。半导体行业每年推出的 MCU 型号多达上千种。以 NXP 一家公司为例，2000 年左右，具备单独订购号码的产品就有 3 万多种。

此外，崇尚快速迭代的互联网企业也为连接技术多样性做出了许多贡献。举个简单的例子，二维码支付抢占 NFC/RFID 的市场份额就是典型案例。在这之前，互联网企业还尝试过其他手

段，如音波支付以及红外支付等。对这种没有条件，创造条件也要上的精神，笔者深表佩服。不过，对于未经系统安全论证就直接上马的做法，笔者则持保留意见。

5.2.1　连接由芯片开始

为了寻找和归类这些连接能力。笔者特地采用高性能 MPU 和入门级 MCU 对比方式，并查看一些综合性半导体公司的产品线分类。这种高低搭配可以填补归类中所产生的盲点，可以寻找到大多数的芯片级基础连接能力。而许多设备级（板级）和系统级连接能力，往往也依赖于这些基础连接能力进行扩展。由于笔者个人从业经验所限，因此许多（仪表、军工、车辆、船舶、楼宇、电讯等）行业的特定连接方式并没有完整统计在内，新型的接口也都没有进行归纳。欢迎读者指正。

Freescale 的 i.MX6 是一款功能强劲的 SoC，其内部功能如图 5-1 所示，该图右侧部分以及中部下方的显示部分接口可以归类为连接能力。可以看到 i.MX6 的连接能力涵盖了移动（嵌入式）和桌面 PC 两大类的各种接口。

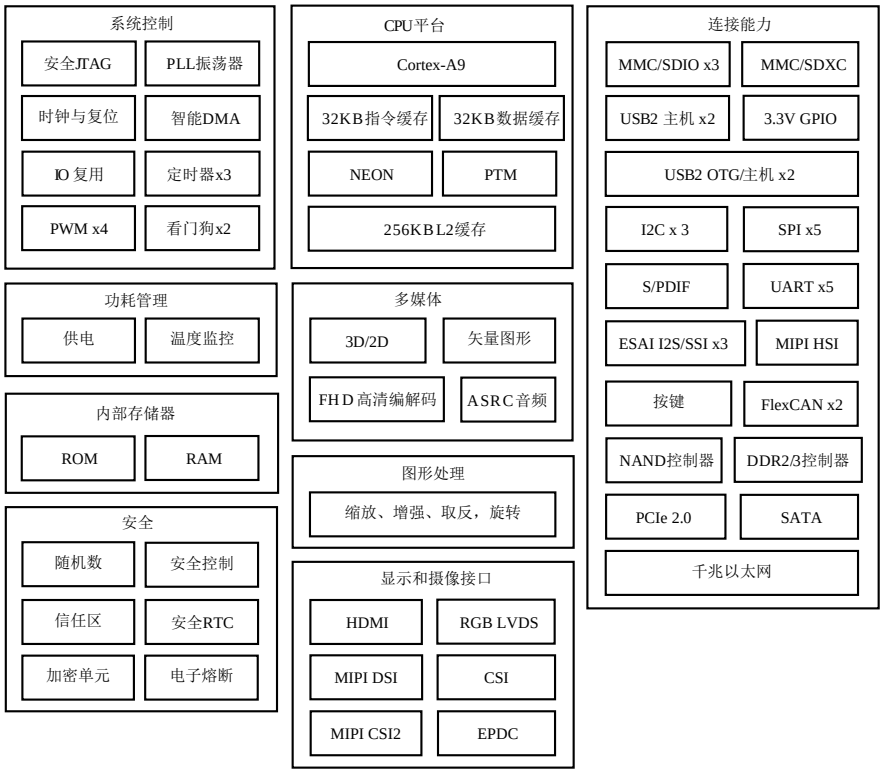


图 5-1　Freescale i.MX6 内部功能框图

- I2C：嵌入式常见接口；
- UART：嵌入式和老式 PC 常见接口；
- CAN：车用网络嵌入式接口；
- MMC/SD：移动设备常见存储器接口；
- ESAI/I2S：嵌入式和 PC 光纤音频接口；
- MIPI：移动设备摄像头接口；
- PCIe：PC 外扩接口，可接 Wi-Fi/3G/4G MODEM；
- SATA：PC 硬盘和 SSD 接口；
- Ethernet（以太网）：PC 常见网络接口；
- USB OTG/Host：移动和 PC 常见外扩接口；
- NAND：闪存原生接口；
- DDR2/3：常见 DRAM 接口。

液晶/OLED 点阵显示屏和摄像头接口也可以归类于物理接口。在许多互联网创新应用中，利用点阵显示屏对外提供二维码显示，用户使用手机摄像头识别二维码获取网址和信息，已经成为一种非常主流的应用方式。在此类情况下，可以将这两者归类于新型连接技术。

图 5-2 是 Freescale 面对计量表行业而设计的 M 系列微控制器（MCU），从图 5-2 中可以看出，其外部连接能力和 i.MX6 有较大区别。

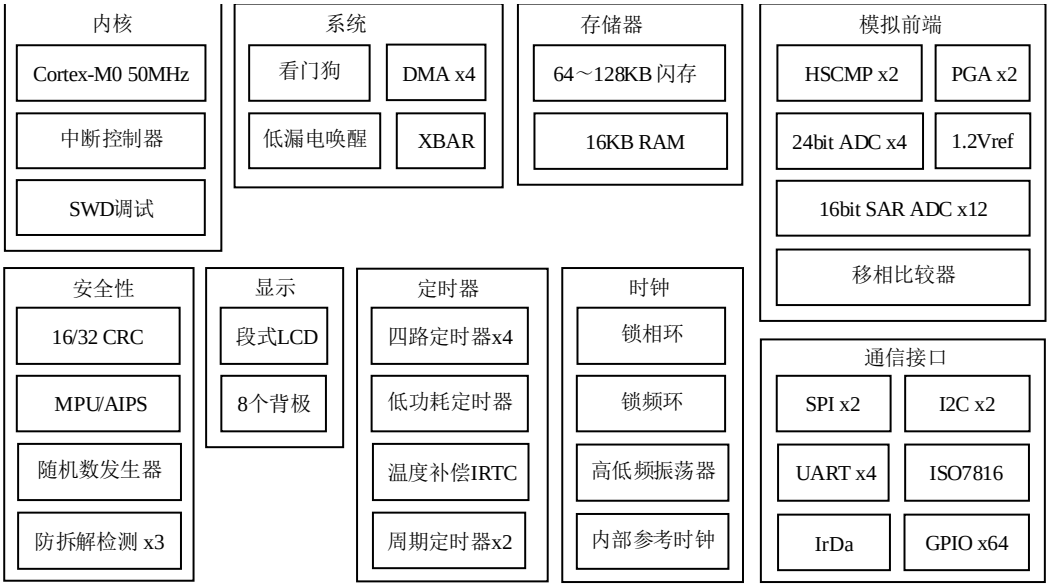


图 5-2　Freescale Kinetis-M 系列微控制器

通信接口：

- SPI，用于 WSN 前端连接；
- I2C，用于系统存储器和实时时钟；
- UART，用于连接其他设备，包括 ISO7816 智能卡 UART，以及红外版本 UART；
- 模拟前端是计量行业专用版本。

模拟相关电路如 ADC/DAC/PWM 等是否属于连接性？一般来说，这是单独分开的。模拟电路一般用于数据采集。但这些模拟相关电路，在许多特殊领域也有着与外界通信的能力。比如车联网中 OBD 所支持的协议除了 CAN 总线，还有利用脉冲宽度来传递数据的 PWM/VPM 方式。因此，模拟电路的某些场合也可以提供某种连接能力。

I/O 连接能力实际上没有一个准确的划分方法。我们姑且按照芯片、电路板、设备、系统的分级定义来划分不同层次的连接能力：

- 芯片间，同一 PCB 上不同芯片间的连接能力；
- 板间，同一设备中不同 PCB 间的连接能力；
- 设备间，同一系统中不同设备间的连接能力。

芯片、电路板和设备之间的界限许多时候没有那么明确。某些技术可以在不同层次上使用。

5.2.2　芯片内部系统总线

工业界最初的计算机控制板是以 Z80 为代表的单板机（Single Board Computer）。所有关键功能模块如 CPU、SRAM、EPROM、ADC、PWM、Timer 等封装在单独的芯片中。彼此通过锁存器、开关、移位寄存器等耦合逻辑芯片，与数据总线、地址总线等并行总线相连。在此基础上，产生了 Z80/8051/68000 等体系的总线接口。

SBC 时代后是 MCU 时代（微控制器，Micro Controller Unit；又称单片机，Single Chip Controller）。这个阶段工业界的主导架构为 8051 以及各类私有指令架构。其与前一代的最大区别在于 CPU/SRAM/ROM 等功能模块整合设计在一枚芯片中。除了为 RAM 扩展而保留的总线，更多的 MCU 品种都将原有并行总线封装在片内。大多数周边设备被整合设计为片内外设（On-Chip Peripheral）。

SoC（System on Chip，系统芯片）除了日益整合的片内外设，多采用 ARM/MIPS 内核。其内部出现了流水线、多核、大小核、总线分级、DMA 等日益复杂的技术。这些底层硬件细节，往往与高级编程语言无关。不过固件开发者依然需要了解某些硬件细节，否则容易出现各种故障和应用问题。

AMBA 片上总线

AMBA 2.0（Advanced Microcontroller Bus Architecture）规范包括四个部分：AHB、ASB、APB 和 Test Methodology。AHB 的相互连接采用了传统的主模块和从模块的共享总线，实现接口与互连功能分离。这对芯片上模块之间的互连具有重要意义。AMBA 已不仅是一种总线，更是一种带有接口模块的互连体系。

AMBA 2.0 规范中的 AHB（Advanced High Performance Bus）是高级高性能总线，包含多个主机、从机、一个总线仲裁器和一个中央译码器。挂在这个总线上面的都是高速模块，如处理器、ROM、RAM、DMA，以及 32 位地址总线和 32 位数据总线。

ASB（Advanced System Bus）总线适用于连接高性能的系统模块。它的读/写数据总线采用的是同一条双向数据总线，可以在某些高速且不必要使用 AHB 总线的场合作为系统总线，可以支持处理器、片上存储器和片外处理器接口及与低功耗外部宏单元之间的连接。

APB（Advanced Peripheral Bus）是高级外设总线，在该总线上面有 USART、GPIO、AD/DA 等外设，通过一个总线桥与 APB 或者 ASB 连接。它不需要很高的时钟频率就可以工作，功耗很低。

半导体供应商设计 ARM SoC 时，基本工作就是将高速模块挂接到 AHB 上，并将各类外设挂架到 APB 上。这里比拼的不再是内核，而是存储器的配置和外设的种类了。

5.2.3 芯片间连接技术

地址和数据总线是并行接口，会占用大量的引脚。在单片机时代，大多数控制器的功能模块，如 CPU、RAM、ROM、ADC、PWM、Timer 都被集成到内部。只有在扩展模式下，RAM（SRAM/DRAM/PSRAM）与内存映射型芯片（NOR Flash 或部分以太网连接芯片）保留了和单片机数据与地址并行总线连接的能力。其他大部分芯片，包括 Flash ROM 均通过 SPI 总线连接到控制器。甚至越来越多的外部设备也被一一集成到了单片机内部，成为片内外设（On-Chip Peripheral）。没有集成在片内的外设（Off-Chip Peripheral）则使用某种串行通信方式连接到控制器。

5.2.3.1 mikroBUS 典型案例

位于塞尔维亚贝尔格莱德市的 MikroElektronika 公司，一直针对教育培训机构以及创客、极客群体提供各类嵌入式开发工具。其定义的 mikroBUS 类似于开源硬件 Arduino 的 Shield，是一种用于连接 MCU 和外部扩展电路板的机械及电气结构。图 5-3 即 mikroBUS 接口示意图，请参阅表 5-1 对照、了解引脚功能。

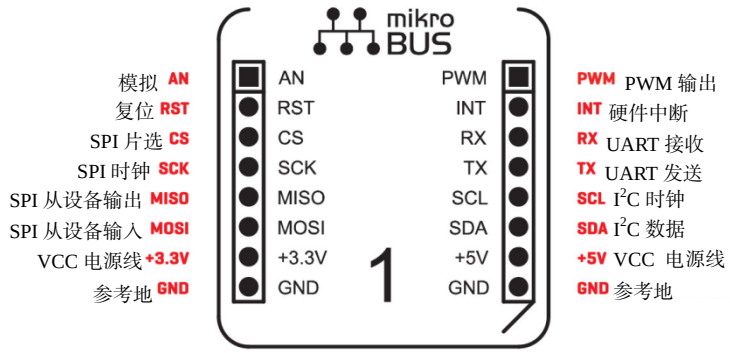


图 5-3　mikroBUS 接口示意图

mikroBUS 与 Arduino 相比，只提供了一组最基本的 I/O，接口非常精简。但其一般不支持堆叠安装模块，而是并排安装多个模块。

表 5-1　mikroBUS 接口定义

名　称	功　能	方　向
AN	ADC 模拟输入	Shield→MCU
RST	复位	MCU→Shield
CS	SPI 片选	MCU→Shield
SCK	SPI 时钟	MCU→Shield
MISO	SPI 数据线，主控输入从控输出	Shield→MCU
MOSI	SPI 数据线，主控输出从控输入	MCU→Shield
3V3	3.3V 供电，<500mA	主板 LDO 提供
GND	接地，参考地	
PWM	PWM 输出	MCU→Shield
INT	硬件中断输出	Shield→MCU
RX	UART 接收	Shield→MCU
TX	UART 发送	MCU→Shield
SCL	I2C 时钟线	MCU→Shield
SDA	I2C 数据线	双向
5V	5V 供电	主板 USB 提供

注意　mikroBUS 并没有规定电气连接方向。实际上，mikroBUS 的一些扩展板的许多 I/O 也是复用的。读者在使用时需要参考具体扩展板规格书。“方向”一栏是笔者通过对比不同扩展板后得出的结论，仅供参考。

mikorBUS 的 5V/3V3 供电分别由主控板的 USB VBUS 和板载 LDO 提供。但在基于锂电池供电的设计中，供电设计会变得更复杂一些。

mikroBUS 接口定义非常简单。模拟接口只有 ADC/PWM，数字接口只有 SPI/I2C/UART。在 MikroElektronika 的推广下，已经积累了 200 多款自有品牌 Click 扩展板。而且多个 mikroBUS 开发板可以组合，构成完整应用生态系统。

mikroBUS 接口证明了一点：芯片级连接能力虽然种类不多，基础连接种类却已经高度集中统一到少数几种接口。

5.2.3.2 数字连接

表 5-2 罗列了一些常见芯片级数字连接方式。

表 5-2 芯片级数字连接方式

数字接口	同步方式	经典速率	CPython	嵌入式 Python
UART/USART	异步	300bps~2Mbps	pyserial	C 扩展
I2C	同步	最高 3.4Mbps	python-smbus	C 扩展
SPI	同步	<20Mbps	python-spidev	C 扩展
1-wire	自同步	<20kbps	N/A	C 扩展
ISO7816	同步	>9.6kbps	N/A	C 扩展
USB Device	自同步	12Mbps~480Mbps	N/A	C 扩展
USB Host/OTG	自同步	480Mbps	PyUSB	C 扩展
CAN	自同步	1Mbps	N/A	C 扩展
Ethernet	自同步	100Mbps-1Gbps	socket	C 扩展
LCD/VFD/EPD	同步	<1Mbps	N/A	C 扩展
QEI 编码器接口	同步	N/A	N/A	C 扩展

5.2.3.3 模拟连接

前面提到过模拟连接能力在某些场景中可以作为系统与外部的连接手段，在此有必要整理一下。表 5-3 罗列了一些常见的芯片级模拟连接方式。

表 5-3 芯片级模拟连接方式

模拟接口	主要用途	应　　用	CPython	嵌入式 Python
模拟比较器	与参考电压比较	物理量输入	N/A	C 扩展
ADC	模拟输入	物理量输入	N/A	C 扩展
DAC	模拟输出	物理量输出	N/A	C 扩展
PWM/SCT	电机驱动	电机/音量	N/A	C 扩展
CapSense	电容接口	人机界面	系统坐标	C 扩展

5.2.4　设备间连接

表 5-4 列出了部分板级的连接接口，其中部分已经被整合进 MCU/MPU，比如 SDCard/MMCard/SDIO 就是如此。

表 5-4　板级（间）连接

名　　称	用　　途	速　　率	Python
ISA	I/O 扩展	16MB/s	ctypes
AGP	显卡局部总线	580MB~2.1GB/s	ctypes
PCI	I/O 扩展	133MB~1GB/s	ctypes
PCI Express	I/O 扩展	250MB~8GB/s	ctypes
VXI/PXI	仪表 I/O 板卡	133~528MB/s	ctypes
STD	Z80 工业板卡		ctypes
IDE/PATA	存储扩展	66~133MB/s	文件系统
SATA	存储扩展	150~600MB/s	文件系统
SCSI	存储扩展	3~640MB/s	文件系统
CFCard	存储，I/O 扩展	16~167MB/s	文件/ctypes
SDCard	存储，I/O 扩展	10MB	文件/ctypes
MMCard	存储，I/O 扩展	10MB	文件/ctypes
PCMCIA/CardBus	存储，I/O 扩展	90MB	文件/ctypes
ExpressCard	存储，I/O 扩展，带 USB 主机	PCI/USB 速率	多种方式

设备间的连接，包括传统的 UART 和 RS232/485 等。不过 UART 连接不仅可用于设备通信，也可用于芯片间通信。表 5-5 列出了部分设备间的连接技术。

表 5-5　设备级（间）连接技术

名　　称	底层连接方式	速　　率	Python
IrDA	UART	<16Mbps	pyserial
RS232	UART	256kbps	pyserial
RS422	UART	<10Mbps	pyserial
RS485	UART	<10Mbps	pyserial
Access.BUS	I2C	<3.4Mbps	C 扩展
IEEE 488/GPIB	双向并行接口	<1MBps	C 扩展
VXI/PXI	PCI	528MB/s	N/A
PS/2	SPI 类似	15kbps	标准输入

续表

名　　称	底层连接方式	速　　率	Python
USB	USB	480Mbps	PyUSB
IEEE 802.3	Ethernet	1000Mbps	socket

表 5-6 中列出了部分音视频专用目的的连接技术。这往往和 Python 没有特别的联系，但却可以通过其他系统服务进行访问。

表 5-6　音视频连接技术

名　　称	底层连接	速　　率	Python
VGA	带 I2C	EDID	模拟信号
HDMI/DP/DVI	带 I2C	EDID	N/A
SPDIF/I2S	特殊时序	音频播放	N/A
1394 Fireware	内部桥接	文件系统或 socket	file/socket
CSI/DSI	特殊时序	video/LCD 面板	N/A

5.2.5　设备组网

组网技术就是计算机网络组建技术。计算机网络的类型有很多，根据不同的组网技术有不同的分类依据。按拓扑结构可分为总线型、星型、环型、树型、全网状和部分网状网络。按传输介质又可分有线网络和无线网络。有线网络是指采用同轴电缆、双绞线、光纤等有线介质连成的网络。无线网络是指采用电磁波作为载体来实现数据传输的网络类型。根据网络分布规模来划分的网络为局域网（LAN）、城域网（MAN）和广域网（WAN）。实际上，行业内还有针对更细分的 PAN/BAN 等划分。

5.2.5.1　OSI 七层模型

涉及设备组网和设备联网，必须提到 ISO 的 OSI 模型，即国际标准化组织的开放系统互联模型，共分七层，具体定义可参阅表 5-7。

表 5-7　ISO OSI 模型

层数	英　　文	中　　文	功　　能	例　　子
7	Application	应用层	应用程序	WWW、SMTP、FTP 等
6	Presentation	表达层	代码格式，加密、解密，压缩、解压缩	
5	Session	会话层	身份验证、会话管理	
4	Transport	传输层	流量控制和负载均衡	TCP/UDP/TLS
3	Network	网络层	网络连接方式和路由选择，虚电路或数据报	IP 地址和端口

续表

层数	英　文	中　文	功　能	例　子
2	Data Link	数据链路层	介质访问 MAC，逻辑链路 LLC，顺序/流量/错误控制	PPP/HDLC，802.3/802.5/CSMA/CD
1	Physical	物理层	与硬件有关联的机械和电气特性	RS232/RS485

从功能角度可分为三组，1、2 层解决网络信道问题，3、4 层解决传输问题，5、6、7 层处理对应用进程的访问。从控制角度可分为两组，第 1、2、3 层是通信子网层，第 4、5、6、7 层是主机控制层。

即便是不开放系统，现在许多系统也会参考 OSI 模型进行设计实施。但其层次会做简化甚至合并，与标准七层模型有差异。

1．传输层上的虚拟物理层

比如嵌入式系统里最常见的 UART/RS232 通信，原始串口通信基于物理层。因为是点对点通信，所以其大部分仅有物理层、中间层和应用层。而且中间层和应用层也是混用的。现在出现了各类虚拟串口：基于 USB/蓝牙/红外/TCP 等各类传输层。那么原有的串口协议则被视为会话层、表达层和应用层。

2．PHY/MAC 软硬件实现

通常物理层与软件无关。但在某些场景中，有用软件实现低速物理层的设计，比如软件 UART、软件 USB、软件 I2C、软件 SPI 等。此外，虽然软件可以实现 MAC 层，但出于性能的考虑，用硬件实现 MAC 也非常常见，以太网和 RFIC 都是如此。各类无线协议的设计重点在于如何根据应用需求，设计实施数据链路层中的 MAC/LLC/网络层。

3．PHY/MAC 共享

在各类系统中，经常出现共享底层，但是网络层和传输层不同的设计。典型例子如下：

- 基于同轴电缆物理层出现以太网、Cable MODEM 等不同设计；
- 基于 IEEE 802.15.4 出现 Zigbee、6LowPAN 和 Wireless-HART 等不同的高层协议；
- 基于 CAN 总线的 PHY/MAC 之上出现 J1939/NMEA2000/DeviceNet/OpenCAN 等不同的高层协议。

说了这么多与 ISO/OSI 模型有关联的事情，笔者想表达的意思如下：

- 请参考模型做设计，这是简化编程、理清思路的方式之一。
- 请灵活掌握原则，必要时精简、采用不同实现方法，不要拘泥于该模型。
- 系统扩展性依赖于分层模型。系统都将逐渐向 TCP/IP 过渡。

5.2.5.2　工业连接技术

工业（含农业）领域的连接技术种类非常多，这里只是罗列一下最常见的几类（参见表 5-8

所列）。

表 5-8　工业用连接以及现场总线

名　　称	目　　的	物理层	底　　层	速　　率	拓　　扑	Python
CAN	通用/车用	双绞线/电缆/光纤	SPI/内部总线	1Mbps	无主从抢占式总线	socket
Flexray	车用	双绞线/光纤	MLI/内部总线	20Mbps	总线/星型	N/A
LIN	车用	12V 信号/同步线	UART	20kbps	单主多从星型	pyserial
D2B/MOST	车用媒体	光纤	内部总线	25Mbps	环型	N/A
IEBus	车用媒体	双绞线	内部总线	50Mbps	总线	N/A
J1939	重型车用	双绞线/光纤	CAN	1Mbps	总线	socket
K-LINE	车用	单端传输	UART	10400bps	单主多从	pyserial
KWP	车用诊断		K-LINE/CAN			socket/pyserial
NMEA	船用	双绞线/光纤	CAN/J1939	1Mbps	总线	socket
电流环	仪表变送器	双绞线	AD/DA	N/A	点对点	N/A
RS485	通用	双绞线	UART	10Mbps	星型	pyserial
ModBus	通用	双绞线/光纤/无线	UART	10Mbps	单主多从星型	pyserial/socket
M-Bus	建筑与计量仪表	双绞线，下行电压/上行电流	UART/AD	9600bps	总线	pyserial
DeviceNet	低压开关控制	CAN	双绞线等	512kbps	点对点/主从/多主	socket
ProfitBus	通用	双绞线/光纤	RS485/IEC1158	12Mbps	令牌环	pyserial/socket
BACnet	楼宇自动化	多种	以太网/RS485/RS232/LonTalk	多种	多种	socket/pyserial
C-Bus	楼宇与照明自动化	双绞线	以太网	CSMA/CD		socket
DALI	照明控制	单端	定时帧结构	1200bps	主从	GPIO/socket
DMX512	照明控制	双绞线	RS485	250kbps	主从星型	pyserial
KNX	家庭/楼宇自动化	双绞线/电力线/以太网/RF	多种	多种	总线型、树型、星型	socket

续表

名　　称	目　　的	物理层	底　　层	速　　率	拓　　扑	Python
LonWorks	楼宇自动化	多种	多种	多种	多种	socket
X10	家庭自动化	电力线	RS232	19.2kbps	总线型	pyserial
ControlNet	工业/通用	同轴/光纤	以太网		总线型、树型、星型	socket
HART	工业仪表	双绞线电流环	电流环+音频 FSK	1200bps	主从	socket
InterBus	工业仪表	双绞线/光缆/红外	以太网		总线	socket
EtherCAT		双绞线	以太网		总线	socket
CC-link	工业控制	双绞线	RS485	10Mbps	主从与点对点	pyserial
FF	工业控制	双绞线/同轴/光缆/RF	令牌总线	总线/树型/菊花链	socket/pyserial	
Dupline	建筑/给水/能源/铁路	双绞线/光缆/RF/GSM	串口	多种	多种	pyserial/socket

这里仅简单罗列了一些工业、汽车、楼宇自动化、智能电网的连接技术。更多连接技术请读者参阅维基百科或百度百科内容。之所以出现各种各样的接口和总线，和各个行业的准入壁垒、市场条块分割有着密切联系。工业控制网络的进展远不及商业网络，原因与标准太多、特殊的应用与技术特点、工业控制领域网络化程度不高有密切关联。商用或民用网络虽然还不断出现新标准，但是大体上呈现出开放和统一的趋势。各个行业的私有连接标准必须采用网关来实现与公共 IP 网络对接。无论是私有协议还是开放标准，协议的 TCP/IP 化是一个总的技术趋势。

现场总线技术在历经了群雄并起、分散割据的初始阶段后，现在的标准是 IEC 61158。其中有较强实力和影响的有 Foundation Fieldbus（FF）、LonWorks、Profibus、HART、CAN、Dupline 等。它们具有各自的特色，在不同的市场和应用领域形成了各自的优势。

此外，物联网的三层模型在这些领域中也有对应的版本，因为传感器、现场控制与远程控制的技术需求是不一样的。Rockwell 提出了 DeviceNet/ControlNet/EtherNet 三级网络对应。其他厂家也有对应的方案。

除了这些有线连接，工业界也逐渐开始向 WSN 网络迁移。当然了，工业无线通信协议的多样化和有线连接协议一脉相承，其依然处于多标准混战的阶段。

5.2.6　设备组网与联网的无线技术

设备组网和联网采用无线技术已经逐渐成为技术趋势。表 5-9 中列出了在消费电子和 WSN

无线传感器网络中的常见方案。

表 5-9　无线连接技术（消费电子及 WSN 无线传感器网络）

名　　称	频　　率	速　　率	拓　　扑
IEEE 802.15.4	2.4GHz	250kbps	点对点/星型
Zigbee	2.4GHz	250kbps	星型/树型/网状网
RF4CE	2.4GHz	250kbps	点对点
6LowPAN	2.4GHz	250kbps	星型/树型/网状网
WirelessHART	2.4GHz	250kbps	网状网
Wirelss M-bus	169/433/868MHz	多种	星型
Z-Wave	862/908MHz	9.6kbps	树型/网状
ANT+	2.4GHz	1Mbps	星型
Bluetooth	2.4GHz	2.1Mbps	星型/网状网
IEEE 802.11	2.4/5.8GHz	600Mbps	星型
RFID/NFC	13.56MHz	424kbps	点对点
NFMI	13.56MHz	424kbps	点对点
Wireless USB	2.4GHz	480Mbps	星型
Wireless A/V	2.4/5.8GHz	与调制方式有关	星型/点对点

由于各种（政治、经济和历史）因素的综合作用，在全球各地的无线电频谱划分存在很大差异。也就是说，在无线频谱的使用上，存在着空间和频率的划分。由此产生了许多附加的技术和商业问题。国通信联盟无线电通信局（ITU-R）规定了 ISM 频段（Industry，Science，Medical），即工业、科学和医疗应用频段可有限度地免费使用部分无线电频谱。国际通用的 ISM 频段集中在 2.4GHz/5.8GHz，发射功率必须低于 1W，且不得对其他设备造成干扰。

5.2.6.1　2.4GHz 与 SubGHz 无线技术对比

目前困扰物联网行业的一个致命问题就是同频干扰与邻频干扰太多。433MHz/2.4GHz 是全球通用的，对应干扰也多。除了 2.4GHz 频段，北美地区的 902～928 MHz、欧洲的 863～870 MHz、中国新开放的 779～787 MHz 以及 470～517MHz，也属于无须申请许可证就可以使用的频段。

ISM 频段中 2.4GHz 的频率特性有些特殊。家用微波炉充分利用了 2.4GHz 的频率特性对食品，尤其是含水的食品进行加热。2.4GHz 在湿润环境中衰减快，波长短绕射特性差。同时由于 ISM 频段的规定，发射功率受到限制，致使它传输距离较短，建筑与金属穿透能力差，而且在野外和湿润环境中距离衰减得更快。在中国，许多公司曾经尝试利用 2.4GHz Zigbee 用于自动抄表，但却出现了各种问题。所以，中国专为 IEEE 802.15.4C 标准预留了 779~787MHz 频段，这同时也是为物联网行业发展预留的免费使用频段。

100MHz~1GHz 之间的频率，在通信领域被统称为 SubGHz。全球范围内的物联网频谱利用趋势是采用 SubGHz 的版本替代 2.4GHz 版本。6LowPAN、Zigbee、Wi-Fi 等，都有从 2.4GHz 向 SubGHz 迁移的趋势，出现了 SubGHz 的版本。行业最佳实践结果就是启用 SubGHz 频段。所以，新开发的物联网项目必须留意这个变化。2.4GHz 相对于 SubGHz 的优势是可以用于较为精确地 ToF 定位，可以用于资产与人员定位。

随着多种 2.4GHz/SubGHz ISM 频段无线技术的采用，物联网出现了一些新的技术趋势：首先，许多半导体厂商推出了整合型多模多协议的 RFIC/SoC，进行无线技术整合：有的整合 Wi-Fi/BLE，有的整合 802.15.4/BLE/ANT/NFC，有的整合 SubGHz 与 802.15.4。读者需要留心观察此类供应商的动向。

除了物理层集成电路级别的整合，网络层与应用层的整合也在发展。Zigbee 最大的应用问题就是互操作性，而 Google Thread 的使用，以及 Zigbee dotdot 语言的推出正是用于解决此类问题以及设备间协同的。6LowPAN/Thread 在 BLE5 中的应用也说明 Mesh/IPv6 已经逐渐在消费场景中普及。

5.2.6.2　IEEE 802.15.4

LR-WPAN（低速率无线个人网络）是一种结构简单、成本低廉的无线通信网络，它使得在低功耗和低吞吐量的应用环境中使用无线连接成为可能。与 WLAN 相比，LR-WPAN 网络只需很少的基础设施，甚至不需要基础设施。IEEE 802.15.4 工作组的任务就是定义 LR-WPAN 的标准与协议。

从图 5-4 的 IEEE 802.15.4 协议分层示意图中可以看出：

- 高层包括网络传输层和应用层；
- 应用层提供设备的既定功能；
- 网络层提供网络配置，操作信息路由；
- 网络层可以直接访问 MAC 层服务；
- IEEE 802.2 为逻辑链路层（LLC），通过服务协议汇聚子层（SSCS）访问 MAC 层服务；
- IEEE 802.15.4 为 LR-WPAN 应用制定了物理层（PHY）和媒体接入控制层（MAC，或称媒体访问层）协议。

IEEE 802.15.4 是 Zigbee、RF4CE、6LowPAN、Wireless-HART 的底层协议。Microchip 的 MiWi 产品线，也在 802.15.4 上构建自己的高层堆栈。

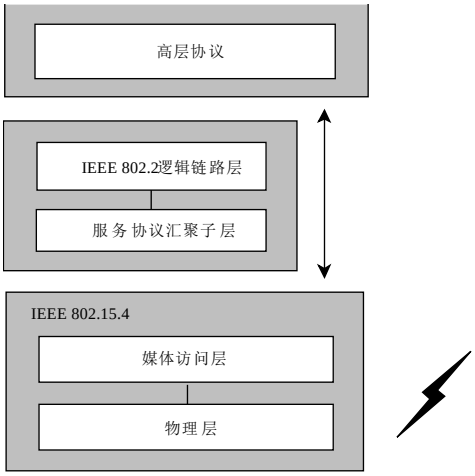


图 5-4　IEEE 802.15.4 协议分层示意图

1．IEEE 802.15.4 协议栈

IEEE 802.15.4 标准定义的 LR-WPAN 网络具有如下特点：

- 在不同的载波频率下实现了 20kbps、40kbps 和 250kbps 三种不同的传输速率；
- 支持星型和点对点两种网络拓扑结构；
- 有 16 位和 64 位两种地址格式，其中 64 位地址是全球唯一的扩展地址；
- 支持 CSMA/CA（Carrier Sense Multiple Access with Collision Avoidance），即采用避免冲突方法的载波监听多路访问；
- 支持确认（ACK）机制，保证传输可靠性。

2．IEEE 802.15.4 的版本

- IEEE 802.15.4 – 2003：最初版本，提供了两种不同的 PHY 设计，一种是针对 868/915MHz SubGHz 频段，一种是针对 2.4GHz。
- IEEE 802.15.4 – 2006：2006 版提供了在更窄频宽基础上的更高速率。其对 868/915MHz 频段 PHY 进行了更新；同时还定义了四种调制技术，其中三种为 SubGHz，一种为 2.4GHz。
- IEEE 802.15.4a：该版本定义了两种新的 PHY，一种使用 UWB，另外一种在 2.4GHz 上使用 chrip 扩频技术。（UWB，即 Ultra Wideband，这是一种无线载波通信技术，通过对具备陡峭跳变的冲击脉冲进行直接调制，信号具备 GHz 量级带宽。）
- IEEE 802.15.4c：更新了 2.4 GHz、868 MHz 和 915 MHz、UWB 和中国 779-787 MHz 频段。
- IEEE 802.15.4d：2.4 GHz、868 MHz、915 MHz 及日本 950～956 MHz 频段。
- IEEE 802.15.4e：该版本定义了增强型 MAC 实现，让 IEEE 802.15.4 in 支持 ISA

SP100.11a 应用。

- IEEE 802.15.4f：该版本定义了 UWB 的 PHY、2.4GHz 以及 433MHz 频段。
- IEEE 802.15.4g：定义了智能相邻网络的 PHY 实现，可用于智能电网，包含 902～928 MHz 频段。

5.2.6.3　6LowPAN

6LowPAN 是 IPv6 over LowPAN 的缩写，最初的 MAC/PHY 只有 IEEE 802.15.4，包括 868MHz/915MHz/2.4GHz 三个载波版本。不过随着多种无线技术的使用，6LowPAN 还可以基于 BLE、Wi-Fi 和 SubGHz 无线实现。SubGHz 的功率和频率特性与 2.4GHz 不同。其更加适合在复杂和野外环境中使用。但目前没有全球标准化的 SubGHz 6LowPAN MAC/PHY 设计。

IETF 6LowPAN 工作组的任务是定义在如何利用 IEEE 802.15.4 链路支持 IP 通信的同时，遵守开放标准以及保证与其他 IP 设备的互操作性。这可以消除对多种复杂网关、专用适配器、专有安全与管理程序的需求。6LowPAN 工作组发明了一种将 IP 包头压缩到只传送必要内容的小数据包中的方法。这些方法去除 IP 包头中的冗余或不必要的网络级信息。IP 包头在接收时从链路级 802.15.4 包头的相关域中得到这些网络级信息。为了与嵌入式网络之外的设备通信，6LowPAN 增加了更大的 IP 地址。当交换的数据量小到可以放到基本包中时，可以在没有开销的情况下打包传送。对于大型传输，6LowPAN 增加分段包头来跟踪信息如何被拆分到不同段中。

开源 6LowPAN 设计了 Conkiti、TinyOS，并部署在了许多工程中。读者在研发自己的堆栈中，也可以参考这些开源设计。将 IPv6 导入后，可以大大简化高层的设计应用。

5.2.6.4　5.8GHz 连接技术

5.8GHz 连接技术多采用正交频分复用（OFDM）调制方式，采用点对点、点对多点组网方式。其虽然也属于 ISM 频段，但是频谱开放时间短，所以同频干扰和邻频干扰不太严重。

- 5.15～5.25GHz，规定其 EIRP 不大于 23dBm，适用于室内无线通信。
- 5.25～5.35GHz，规定其 EIRP 不大于 30dBm，适用于中等距离通信。
- 5.725～5.825GHz，通常人们选用 5.725～5.825GHz 来进行社区的宽带无线接入，以获得更高的性价比。

EIRP（Effective Isotropic Radiated Power）有效全向辐射功率，也被称为等效全向辐射功率（Equivalent Isotropic Radiated Power）。

5.8GHz 的产品标准有 802.11a、FCC Part 15、ETSI EN 301 489、ETSI EN 301 893、EN 50385、EN 60950 等。目前 5.8GHz 的应用主要如下：

- 宽带无线接入；
- 无绳电话；
- 网吧、话吧接入；

- 综合电信业务接入；
- 基站互联，使用点对点拓扑；
- 高速无线音视频传输；
- 无人机的数据链传输（图传）；
- 车联网与雷达。

5.2.6.5 WAN 连接技术

WAN 连接往往依赖于公众数字移动蜂窝通信网络中的数据传输业务，如下所示。

- 2G：GSM CSD/GPRS/EDGE，CDMA 1X/95；
- 3G：WCDMA（HSDPA/HSUPA/HSPA+），CDMA 2000/EVDO，TDS-CDMA；
- 4G：TD-LTE/FDD/LTE。

此类 WAN 连接技术请参见表 5-10 所列。

语言业务是 2G 移动通信的重点，而 3G/4G 乃至以后的 5G 的发展重点是数据通信。语言业务将成为数据业务中的一部分，比如 VoLTE/VoIP。虽然短消息、彩信曾经也是物联网的基础技术，但目前基本上已经被边缘化了。新的 LPWA 技术如 NB-IoT 和 LoRa 则是针对物联网的特性而设计的长距离低功耗连接技术。其中 LoRa 可以工作于 ISM 频段，无须申请使用许可。

表 5-10 WAN 连接技术

名　　称	上行速率	下行速率
GSM/GPRS/EDGE	53.6kbps~171.2kbps	384kbps
WCDMA/HSDPA	5.76Mbps	14.4Mbps
FDD-LTE	40Mbps	150Mbps
TDS-CDMA/HSDPA	2.2Mbps	2.8Mbps
TD-LTE	50Mbps	100Mbps
CDMA95/1X	153kbps	153kbps
CDMA2000/EVDO	1.8Mbps	3.1Mbps
NB-IoT	64kbps	28kbps
eMTC	1Mbps	1Mbps
LoRaWAN	300bps~37.5kbps	300bps~37.5kbps

国内的运营商授权频谱划分方案请参阅中国无线电管理委员会官方网站信息。

在许多地广人稀、公众移动通信网络无法覆盖的地方，可以通过报批主管部门安装移动基站铺设专网来解决覆盖问题。此外，其他行业，如广播媒体、卫星通信、铁路等行业也都有各自的 WAN 传输技术标准，这在某些特定应用场景中有着独特的用途。和消费者密切关联的大

概只有 DAB/DMB、FM-RDS 之类与媒体广播有关的技术了。但是国内广电系统的技术没有得到充分的发挥，这是一件很可惜的事情。

5.2.6.6　LoRaWAN

Semtech（升特科技）的 LoRa RFIC 很适合构建低速率超长距离的 LPWAN 专网。不同于现有方案，LoRaWAN 的特点如下：低功耗、传输距离远，可以涵盖 1~10km 距离，采用线性调制技术，低速率情况下最远可达 20km。LoRa 传输速率为从 300bps 到 37.5kbps。速率越低，传输距离越远。低功耗广覆盖的代价是低速率，所以过于复杂的协议比较难以实施。

其低功耗特性可以让节点使用内置电池间歇工作 10 年时间。其覆盖面积广，可以提供数百万网络节点接入；也可以通过不同的基站配置形成类似于蜂窝的设计。目前，美国纽约市、中国台湾省台北市，以及中国大陆都有实验网可供使用。

LoRa 芯片家族常见型号列于表 5-11 中。LoRa 芯片支持多种调制解调技术，包括常见的 OOK 和 FSK、LoRa 线性调制。相同环境条件下，LoRa 传输距离是 FSK 的 1.5~1.8 倍，速率越低，差别越大。

表 5-11　LoRa 集成电路

IC	频　率	市　场
SX1272	860MHz~1GHz	欧洲
SX1273	860MHz~1GHz	欧洲
SX1276	137MHz~1020MHz, 14/20dBm	全球
SX1277	137MHz~1020MHz	全球
SX1278	137MHz~525MHz, 14/20dBm	中国，东南亚，南美，东欧
SX1279	137MHz~960MHz	全球
SX1236	137MHz~1020MHz	全球
SX1238	863MHz～870MHz/902MHz～928MHz, 27dBm	欧洲，美洲
SX1231	290MHz～1000MHz RF 收发前端	基站专用
SX1301	长距离通信数据集中器与网络基带处理器	基站专用

常见型号 SX1272/SX1276/SX1278，无论是分销商还是模块供应商的货源都很充足。此外，模块供应商中也提供内含 MCU 的 LoRa 模块。

LoRa 的现状：

- Semtech 创建的 LoRa 联盟中包括了电信运营商、IT 集成商、设备制造商、模块制造商和半导体供应商；
- ST/Microchip 与 Semtech 合作，目前 STM32 + SX127X 成为行业标准配置；

- IBM 和 Semtech 合作规划了 LoRaWAN 协议；
- Semtech LoRa 芯片较便宜，国内采用 SX1278，北美、西欧采用 SX1272/SX1273，要兼顾多重市场的采用 SX1276；
- Semtech LoRa 网关/基站芯片 SX1301 的相对资料不公开，仅针对大客户，需要和原厂签署 NDA 获取资料和支持；
- LoRa 模块，国产价格在 50 元左右；
- LoRa 开发板，比较贵；
- LoRa 节点设备开源固件较多，已经出现了 MicroPython＋LoRa 的方案；
- LoRa 网关开源固件相对较少，主要受限于 SX1301 资料不全。

图 5-5 是 IBM 苏黎世研究所提供的 LoRaWAN 的 LRSC 架构图。图 5-5 中非常明显地突出了 IBM 所扮演的角色：网络服务器（即交换机）和云计算供应商。

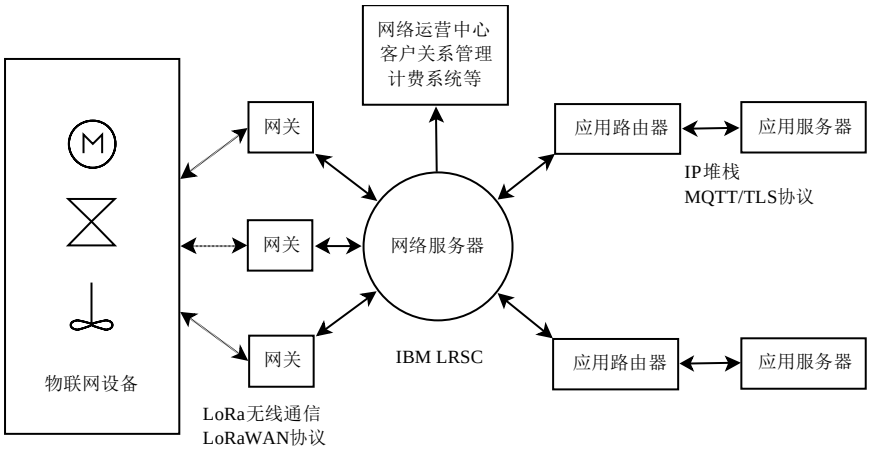


图 5-5　IBM LoRaWAN LRSC 架构图

- 网关：协调通信通道和顺序，控制流量并避免外部信号干扰。
- 网络服务器：管理多台网关并控制上下行通信与数据流、计费、保持端到端的通信加密。
- 应用路由器：与应用服务器交换（MQTT/IPv6）数据。
- 应用服务器：基于 IBM Bluemix 云服务，处理应用业务相关数据。

IBM 对于安全意识很强，从边缘节点到服务器之间均采用 AES 进行加密传输。

LoRaWAN 的网关固件设计相对简单，其内部结构如图 5-6 所示，这里采用了报文转发的方式，将 LoRa 网络中的报文直接发给网络服务器（Network Server），由网络服务器进行处理。网关和云服务器之间采用信号隧道（Backhaul）方式沟通。

信号隧道，又称回程线路，指的是从信源站点向交换机传送语音和数据流量的功能。蜂窝

移动的信号隧道，通过多种物理媒介在基站（BS）和基站控制器（BSC）间建立一个安全可靠电路传输手段。该层传输网络质量直接影响运营商在快速响应业务发展需求、提供可靠的网络业务、降低运维成本和营运业绩等方面的表现。

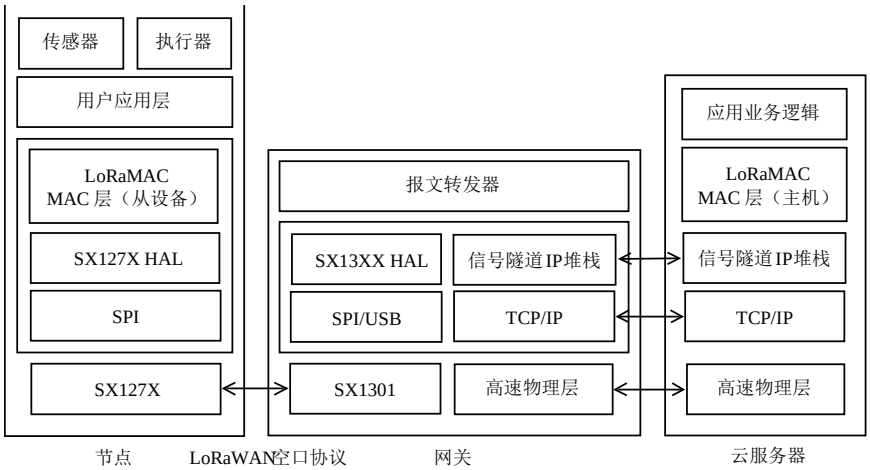


图 5-6　LoRaWAN 网关示意图

在网关框架图中，在 MAC 层使用信号隧道的方式构建网络交换机。信号隧道这个术语来自通信的交换机行业。笔者在查阅资料之后，理解了普通短距无线 WSN 网关和 LoRaWAN 网关之间的差异在于规模和可扩充性。一般的短距离无线电设备，其规模是受控的。但是 LoRaWAN 之类的网络，规模可能不断增加。这就导致了不同的架构需求。从商业角度来看，IBM 在 LoRaWAN 生态链中扮演的就是系统集成角色，所以采用云服务器的网络交换机非常符合其商业利益。

在 IBM 的 LRSC 网络交换机架构中采用信号隧道的设计，网络服务器安装在云端。这种设计可以简化 LoRa 基站的设计要求，可以在云端进行快速迭代，动态调整带宽和拓扑逻辑，增加并行计算能力以支持更多设备接入。但是通过信号隧道将基站 MAC 虚拟化在云服务器中，会需要引入额外的 IP 通信消耗。此类设计需要注意回避拓扑逻辑的无序化，确保信号隧道的安全可靠性，以及尽可能节省带宽成本。

不过 LoRaWAN 毕竟不同于蜂窝通信，其主要使用 ISM 频段，不同垂直应用领域内会出现大量的私有网络与中小运营商，应用规模与公众蜂窝网络相比，相对较小。但是 LoRaWAN 的应用规模跨度可以很大，最小可以是简单的星型拓扑，大型网络可以采用 LoRaWAN 城市网关进行组网。在一些小规模应用中，笔者认为可以采用相对灵活的方式，将微型或者小型网络服务器运行在网关所在的同一计算机或集群内。系统架构不变，保持系统迭代和快速交付能力，但是却可以减少 IP 网络通信消耗。这样可以加快路由器 LoRaWAN 协议的迭代和新增功能的开

发速度。

此外，SX1301 这种网关 IC 仅对供应商认可的客户提供开发资料和技术支持。中小客户可以利用 SX127X 构建星型拓扑网络作为对策。虽然没有 SX1301 可以同时支持多路 DSSS 解调，但是许多中小规模设计是可以使用的。也可以通过并联多个 SX1278 实现多路接收。还可以采用类似于手机基站的蜂窝定向天线，这可以增加覆盖面积内支持的节点数量。

在 GitHub 和 ARM mbed 网站上也有不少相对应的开源固件。读者可以参阅本章延伸阅读部分下载查看。

一个完整的 LoRaWAN 设计需要涵盖硬件、节点固件、网关固件和服务器软件。读者可以先购买已经认证过的网关做开发，也可以设计较为简单的星型拓扑和 MAC 来实现私有网络。不过，LoRaWAN 和各个开源固件可以是一个很好的开端。

5.2.6.7 SIGFOX

SIGFOX 利用 SubGHz 的 ISM 频段设计运行了自己的物联网网络。其底层技术采用 SI446X（SiliconLabs）和 CC112X（Chipcom/TI）收发器，以及 Atmel 的 ATA8520 和 OnSemi/SIGFOX 定制的 AX5043 SoC。SIGFOX 采用 FSK 等较为传统的调制解调方式，传输速率非常低，占空比也非常低，SIGFOX 应该归于私有网络。

5.2.6.8 NB-IoT

2016 年第三季度，3GPP 的 NB-IoT 规格书最后确定。NB-IoT 是 NB-LTE 和 NB-M2M 标准合并的结果。

上海联通已经在浦东金桥做了第一个 NB-IoT 实验网。同时，华为收购了英国 Neul 的 NB-IoT 技术，华为海思推出了 NB-IoT 芯片，瑞士 ublox、上海移远通信，以及 Intel 已经推出了 NB-IoT 模块。这一切说明，NB-IoT 替代原有 2G 蜂窝模块的过程已经开始加速。

NB-IoT 可以基于 GSM/UMTS/LTE 网络进行部署，利用有限的带宽可以实现一个广覆盖、低功耗、低成本的物联网广域网。国内三大运营商都将 NB-IoT 列入了考察目标，都开始建立实验网。2017 年，NB-IoT 模块在 8 美元左右。相信随着规模的扩展，其会迅速向 2 美元目标价位靠拢。

对于之前曾经使用 2G/3G/4G 模块的开发商、运营商、制造商来说，将所有原有设计重新适配到新的平台上不存在特别难度。可以预见的是，会出现大量基于 GPRS/CDMA 的物联网设备的更新抛售潮。

5.2.6.9 LTE 物联网相关标准

除了 NB-IoT，LTE 的各主要供应商还提供了许多其他物联网标准，不过哪一种能够胜出成为主要的标准，尚有待时间检验。

1．LTE Direct

手机可以与其他移动设备以及信标直接通信。这项无线技术被称为 LTE Direct，覆盖范围可达 500m，远大于 Wi-Fi 或蓝牙。

研究人员将 LTE Direct 作为一种允许智能手机自动发现附近人群、企业和其他信息的手段。有人将 LTE Direct 技术视为针对性推广或广告的潜在新渠道。

LTE Direct 的用法非常类似于苹果公司发布的 iBeacon 技术。美国梅西百货公司等零售商将 iBeacon 技术当作一种追踪并连接顾客移动设备的方式，正在对其进行测试。尽管如此，但 iBeacon 设备使用蓝牙协议，覆盖范围要比 LTE Direct 小得多。

理论上，LTE Direct 可以用来开发能在设备之间传送所有数据的通信应用。有些聊天应用已经能够用 Wi-Fi 与蓝牙来连接附近的手机，但 LTE Direct 的覆盖范围有所扩大，性能也更佳。

2．非授权频段 LTE

高通的非授权频段 LTE 是在非授权频段（如 ISM 频段）中使用 4G LTE 的技术。在这些频段上，满足最低功率规格的技术都可以实施。其可以为消费者提供超越 Wi-Fi 的体验，提供更高的吞吐量、更广的覆盖范围和节点容量。非授权频段 LTE 技术分为三种：LTE-U、LAA 和 MuLTEfire，都需要保持与 Wi-Fi 兼容，即与 Wi-Fi 共存。

华为针对非授权 LTE 也有自己的 eLTE-IOT。看来以后针对未授权频段的技术竞争会非常激烈。

3．LTE 广播

LTE 广播可以向同一地区的多个用户传送相同的内容，支持无限量的用户，且能够高效地利用频谱资源和现有网络投资。LTE 广播的优势不止于此。以赛车比赛为例，通过 LTE 广播的多播技术可以在一个平板或者一个手机上有 4 个视角镜头画面，每个赛车上面都装了摄像头，而且每个轨道也装了摄像头，12 个赛车的不同摄像头组成了 4 组同时传输的渠道。

除了多视角播放，还可以在此基础上实现 3D 播放和全景播放等创新播放技术，为 VR 热潮加一把火。

4．LTE MTC/eMTC

LTE MTC（Machine Type Communications）是和 NB-IoT 同时竞争的蜂窝式物联网应用。它适用于那些想要关闭 2G 频段，并将 LTE 网络投放于物联网应用的运营商。但是与 NB-IoT 相比，其功耗依然过高。

LTE MTC 有时候又被叫作 LTE Category 0，该标准中有一部分将纳入 LTE Release 12 规格。其目标是定义一个尺寸较小、较简单的 LTE 模块，售价最好能由目前产品的 25 美元降低至 5 美元。为了达成该目标，未来该规格预期将把最大数据传输速率降低至 2Mbps，并采用众多省电技术。

eMTC（enhanced MTC）与 NB-IoT 经常被拿来比较。NB-IoT 功耗更小；而 eMTC 速率高，

适用于快速移动物联网设备。

5.2.6.10 HaLow 900MHz Wi-Fi

IEEE 802.11ah 现在更名为 HaLow，这是为物联网应用而定制的 Wi-Fi，基于 900MHz。其比 2.4GHz 频段更低，传输速率更低，功耗更低，传输距离更远，且支持的节点数更高。但是其产品化还有很长的路要走。

5.2.6.11 软件无线电

软件无线电，即软件定义无线电（Software Defined Radio，SDR），也就是基于通用无线电收发硬件，利用高速处理器和软件来定义无线电通信的频率、调制方式、空口协议与功能的无线电传播技术。其系统架构通常是 ADC+DSP/CPU+DAC。由于无线技术标准的碎片化，软件无线电在物联网应用中越发重要。

1．GNU Radio

这是开源的软件无线电开发平台。它的开发目的是给软件开发者提供探索无线电的途径和能力。为此，GNU Radio 开发了 USRP/USRP2。GNU Radio 应用采用 Python 开发，其核心 DSP 算法使用 C++，并在浮点运算微处理器上构建。具体教程可以参考本章延伸阅读部分。此外，GNU Radio 可以在没有 RF 硬件的情况下对预先存储的或信号发生器生成的数据进行信号处理算法研究。

2．Realtek SDR

在软件无线电中，Realtek 的 RTL2832U+R820T 组合是入门级的 RF 硬件。其中，RTL2832U 是 USB 2.0 接口的 DVB-T COFDM 解调器，R820T 是硅调谐器。这个配置原来是 DVB-T/DAB/FM 接收器。由于 RF 前端接收范围较宽：24MHz~1766MHz，因此可以作为软件无线电的接收平台。其售价非常便宜，而且支持 GNU Radio。

总的来说，各类开源软件无线电非常适合做接收器。发射无线电需要购买较为昂贵的 PA 和 RFIC，也需要符合当地无线电管理法规。

3．USRP

USRP（Universal Software Radio Peripheral）包含 4 个 64Msps 采样率的 12 位 ADC、4 个 128Msps 采样率的 14 位 ADC，以及 USB 2.0 和收发子板。

4．HackRF

由 Michael Ossmann 发起的开源无线电外设，支持从 30MHz~6GHz，最大带宽为 20MHz。HackRF 在 Kickstarter 上成功实现了众筹。它的特性如下：

- 全面支持 GNU Radio；
- HackRF 不仅仅是接收器，它还可以发射无线电；
- 比 USRP 更加廉价；

- 最大采样率为 20Msps；
- USB 供电；
- 硬件、软件全部开源；
- NXP LPC4330+Xilinux XC264A CPLD；
- MAX2837/MAX5864/RFFC5072 RF IC。

使用软件无线电工具的主要人群为 HAM“火腿族”（也就是业余无线电爱好者）、黑客、网安人员，以及开发者。尤其是预算有限的开发者，买不起昂贵的 RF 频谱分析仪，就可以拿 SDR 来做频谱分析与空口报文分析。廉价的 RTL-SDR 虽然不能发射信号，但是监听信号用于分析却是很有意义的。

5．SDR 产品化

对于通信运营商来说，SDR 基站可以将基础架构升级风险和成本降到最低。由于 3G/4G/5G 的标准之争，因此通信运营商要部署网络的时候应保证与将来标准的兼容性。如果采用基于传统硬件+软件方式大规模升级基站，成本就非常昂贵。而基于高速 ADC/FPGA/CPU 的通用硬件，SDR 基站通过升级软件和部分硬件就可以复用其余部分硬件，从而大大降低成本和风险，加快部署速度。

5.2.6.12　与移动设备相关的连接技术

任何行业都无法忽视移动计算的重要性。笔者曾经以 Android 为主，以 iOS 和其他系统为辅，针对移动设备平台，规划过一系列连接性专题设计。与移动通信设备相关的连接技术参见表 5-12。

表 5-12　与移动通信设备相关的连接技术

连接技术	优　　点	限　　制
移动数据	普及，永久在线，按流量计费	费用较高
USB-OTG	采用 USB Host API，支持用户态驱动	不是所有手机都有 OTG 物理接口
USB Host	USB Host API	API 无法访问系统已经支持的设备
USB-Device	在 MCU 中普及	SoC 可以使用 OTG 替代
USB-ADB	所有 Android 手机都支持	Android 4.0 将 ADB 认证升级为 2048 位 RSA
USB-AOA/ADK	官方推荐外围扩展方式	与传统 USB 设计有差异，较为复杂
USB-C	正反插均可	未普及
Bluetooth	手机均有蓝牙	经典蓝牙设备间存在兼容性问题
Bluetooth Low Energy	新手机都支持 BT/BLE 双模	兼容性弱，不过 BLE 兼容性要优于 Bluetooth 2/3
NFC/RFID	快速数据交换	作用距离短，数据量小，需配合 Bluetooth/Wi-Fi
IrDA	点对点中等速率传输	作用距离短，市场份额小

续表

连接技术	优　点	限　制
Wi-Fi	所有手机都支持	作为 AP 只能够接入三台设备。功耗较高
耳机接口	所有手机都具备，速率低	耳机孔极性兼容性问题
QR 二维码	所有手机都具备	偶有识别问题
声波接口	所有手机都具备	不普及
FM/RDBS	部分手机具备	速率低，单向，文字显示只支持泛欧/日文
闪光灯/摄像头	另类通信手段，成本低	速率低，耗电
振动马达/加速度计	另类通信手段，成本低	速率低，耗电

围绕 Android 手机的连接方式，主流是移动网络+Wi-Fi+BLE，本地连接则采用 OTG/AOA 两种方式，并以摄像头 QR 二维码和 NFC 做辅助配置。

5.2.7　连接性回顾

如此丰富的接口，通过 TCP/IP 网络连接各种云计算、雾计算模式，结合各行各业，物联网的业务和复杂度将以几何级数上升。

罗列了这么多端口、协议、接口，万变不离其宗，最后都会归纳到少数几种基础接口。即使一些少见的物理层接口，也都通过一些桥接 IC，转换成标准端口接入到主控制器中。大部分设计重点集中在协议栈和软件组件，然后构成了如此丰富多彩的应用场景。

C/C++和 Python 可以通过几种主流的通信端口和编程接口访问设备。下面我们重点讲解这些基础通信端口和 Python 在不同操作系统中如何进行对接。由于嵌入式系统大多基于 Linux，因此将是我们介绍的重点。Windows 下主要是三种方式：USB、socket 以及 DLL 对接。

在介绍 Python 与硬件间的接口前，需要先了解操作系统和硬件接口，然后再了解 Python 是如何通过操作系统接口来访问硬件的。了解其原理之后，可以触类旁通，无论硬件如何变化，都可以找到合适的方式来对接软硬件。

5.3　Linux 文件系统

```
l'etat, c'est moi
```

法国国王路易十四曾说道：“朕即国家”（或译为“孤即国家”）。

5.3.1　设备即文件

这里借用上面这句话来表达一个意思：在 Linux 中，包括 I/O 设备在内的对象都被当作文

件来看待。严格地说，Linux 中的大部分设备是文件，但依然有少部分不是文件。

关于 Linux 的设备驱动设计已经超越了本书的范围，在此推荐阅读 O'Reilly 出版社的 *Linux Device Drivers, 3rd Edition*，其中文版名为《Linux 设备驱动程序（第三版）》，书中介绍了 Linux 2.6 的设备驱动设计详情。

Linux 将大部分系统资源以虚拟文件形式呈现给用户。通过标准化的文件 I/O 方式，就可以完成与底层设备之间的数据输入/输出。Linux 文件按照具体对象分类如下。

- 普通文件：一般意义的文件和磁盘文件。
- 设备文件：代表的是系统中的具体设备。
- 管道文件：用于进程间通信。
- 套接字文件：用于网络通信。
- 目录文件：该文件是目录。
- 符号链接：指向另一文件。

5.3.2　设备文件系统

在 Linux 文件系统中，设备文件是一种特殊的虚拟文件，其具体实现方式依赖于特定内核版本。驱动模块可以通过 insmod 连接到正在运行的内核，也可通过 rmmod 从内核中移除。另外一个常用的工具是 modprobe，它会将驱动模块及其依赖的模块一起加载，相当于调用了多次的 insmod。出于安全考虑，设备驱动也可以静态编译到内核中。

Linux 的设备文件大体上分为三种基本类型：字符设备、块设备和网络设备。

5.3.2.1　字符设备

能够按照字节流（类似文件）一样顺序访问的设备，由字符驱动程序实现此种特性。最典型的就是字符终端（/dev/console）和串口设备（/dev/tty*）了。与嵌入式有关的许多外设（I2C/SPI/UART 等）都可以归入此类别。

5.3.2.2　块设备

一次传送 512 或者 2 的倍数字节。虽然从系统调用角度看它和字符设备是类似的，但在内核实现中的内部数据管理和实现则完全不同。这里最典型的就是磁盘设备。

5.3.2.3　网络设备

任何网络通信都通过网络接口，形成一个能够与其他外界交换数据的设备。由于网络底层往往围绕数据分组展开，而 TCP/IP 协议却是面向流/数据报的方式，因此与字符设备不同。在 UNIX/Linux 中往往使用单独的名称进行访问（eth0）。但这个名字在文件中不存在对应节点，内核调用的是网络通信函数而非 read/write 函数。

一些工业网络接口如 CAN 总线，最初采用字符设备类型，从 Kernel 2.6.13 开始提供网络接口（SocketCAN：can0）。这个转变读者需要持续关注，因为任何一种物联网接入技术都有可能发生这种从字符设备到网络设备的转变。CAN 总线是一个典型例子。

还有一些驱动程序类型协同内核中的附加层一起工作。最典型的 USB 模块，每种设备对应一种驱动，设备可以是字符设备（USB 串口）、块设备（MSD U 盘）或者网络设备（USB Wi-Fi），也可以是若干种设备的混合体。典型的混合型设备例子：手机，大多具备 MSD/DFU/MODEM……与 USB 相类似的还有蓝牙。

5.3.3 Linux 设备文件的演变

Linux 的设备文件系统已经经过若干代的迭代和演变，开发者在使用时需要留意这些差别。

5.3.3.1 Linux 早期

在 devfs 之前，设备文件由使用者采用 mknod 创建，都挂载在/dev 路径下。这种方式是使用者手工挂接，静态管理。这种手工体力活不适应快速变化，被 devfs 所取代。

5.3.3.2 devfs

从 Linux 2.4 内核开始导入 devfs。devfs 是一个基于内核的动态设备文件系统。devfs 是为了解决 Linux 早期混乱的设备管理而设计的特殊文件系统，挂载点是/dev。图 5-7 显示了树莓派/dev 路径中的设备清单。从图 5-7 的树莓派文件系统截图中，读者可以看到以下各种设备文件：

- i2c-0/1，系统 I2C 总线；
- spidev0.0/0.1，系统 SPI 总线；
- ram*X*，RAM 设备文件；
- tty*X*，串口设备文件；
- random，随机数发生器；
- mmcblk*X*，SD Card 设备文件。

devfs 有以下几个缺点。

- 不确定的设备映射：一些动态插拔的设备如 USB 设备的映射名称可能是不正确的。
- 没有足够的主、辅设备号：两个字节定义主辅设备号，略显不足。
- /dev 目录下的设备不表示实际设备：使用者不知道这台设备是否处于激活状态。
- 命名不够灵活：devfs 的命名机制要求复杂的配置文件和程序。
- 内核内存使用：作为内核驱动模块，特别是挂接大量设备时，占用大量内存。

于是，devfs 在 2.6.13 之后被 sysfs 替代。

```
pi@raspberrypi /dev $ ls
autofs           MAKEDEV          ram2       tty14  tty39  tty63
block            mapper           ram3       tty15  tty4   tty7
btrfs-control    mem              ram4       tty16  tty40  tty8
bus              mmcblk0          ram5       tty17  tty41  tty9
cachefiles       mmcblk0p1        ram6       tty18  tty42  ttyAMA0
char             mmcblk0p2        ram7       tty19  tty43  ttyprintk
console          mmcblk0p3        ram8       tty2   tty44  uinput
cpu_dma_latency  mmcblk0p5        ram9       tty20  tty45  urandom
disk             mmcblk0p6        random     tty21  tty46  vc-cma
fb0              mmcblk0p7        raw        tty22  tty47  vchiq
fd               mmcblk0p8        rfkill     tty23  tty48  vc-mem
full             mmcblk0p9        root       tty24  tty49  vcs
fuse             net              shm        tty25  tty5   vcs1
i2c-0            network_latency  snd        tty26  tty50  vcs2
i2c-1            network_throughput sndstat  tty27  tty51  vcs3
input            null             spidev0.0  tty28  tty52  vcs4
kmsg             ppp              spidev0.1  tty29  tty53  vcs5
log              ptmx             stderr     tty3   tty54  vcs6
loop0            pts              stdin      tty30  tty55  vcsa
loop1            ram0             stdout     tty31  tty56  vcsa1
loop2            ram1             tty        tty32  tty57  vcsa2
loop3            ram10            tty0       tty33  tty58  vcsa3
loop4            ram11            tty1       tty34  tty59  vcsa4
```

图 5-7　树莓派/dev 路径中的设备文件

5.3.3.3　sysfs 与 udev

sysfs 不仅可以把设备和驱动程序从内核空间转到用户空间，也可以用来设置设备和驱动程序。其文件系统挂载点在/sys 目录下，图 5-8 显示了该路径的文件系统。但这不意味着 sysfs 是完全不同于 devfs 的设备文件系统；事实上，sysfs 保留了/dev 挂载点，如图 5-7 所示；增加了/sys 挂载点，如图 5-8 所示。sysfs 使用 udev 用户空间程序来管理/dev 目录树。通过与文件系统分离，sysfs 让使用者可以定制自己的系统，比如创建设备字符连接，改变设备文件属组、权限等。Linux sysfs 设备文件的相关路径如表 5-13 所示。

```
pi@raspberrypi /sys $ clear && ls -al

total 4
dr-xr-xr-x 12 root root    0 Jan  1  1970 .
drwxr-xr-x 23 root root 4096 Jan 19  2016 ..
drwxr-xr-x  2 root root    0 Aug 12 07:29 block
drwxr-xr-x 16 root root    0 Aug 12 07:29 bus
drwxr-xr-x 42 root root    0 Feb 24 18:17 class
drwxr-xr-x  4 root root    0 Aug 12 07:29 dev
drwxr-xr-x  8 root root    0 Feb 24 18:17 devices
drwxr-xr-x  2 root root    0 Aug 12 07:29 firmware
drwxr-xr-x  4 root root    0 Feb 24 18:17 fs
drwxr-xr-x  7 root root    0 Aug 12 07:29 kernel
drwxr-xr-x 72 root root    0 Feb 24 18:17 module
drwxr-xr-x  2 root root    0 Aug 12 07:29 power
pi@raspberrypi /sys $
```

图 5-8　树莓派/sys 路径中的文件列表

表 5-13　Linux sysfs 设备文件的相关路径

路　　径	说　　明
/dev	设备文件存储目录。通过对该路径下设备文件的读/写和控制，访问实际设备
/sys/devices	按照设备挂接的总线类型，组织成层次结构，保存了系统的所有设备信息
/sys/dev	存放的是块设备和字符设备的主辅号码，并指向/sys/devices
/sys/block	系统中所有的块设备
/sys/bus	系统中以及注册的总线类型，与/sys/devices 对应
/sys/class	所有注册在内核中的设备类型，是 Linux 统一设备类型的一部分
/sys/firmware	对于固件对象和属性进行观察和操作的接口
/sys/fs	描述文件系统及挂载点
/sys/kernel	内核可调整参数
/sys/module	记录被系统内核载入的模块
/sys/power	系统电源选项，可以通过写入几个属性文件实现系统关机、重启等
/etc/udev/udev.conf	udev 配置文件
/etc/udev/rules.d/	udev 规则文件

综上所述，/sys/devices 是系统设备信息在内存中的表达，而/sys/dev 提供了主辅号码。udev 作为 sysfs 的一部分，其配置文件位于/etc/udev/udev.conf，该文件指向了具体的 udev 命名规则。udev 会定时扫描/sys/devices 下的挂载情况，并按照用户命名规则来修改/dev 路径下的设备文件。

sysfs 继承了 devfs，但是增加了一部分自动化管理功能。使用 udev 后，在/dev 目录下就只包含系统中真正存在的设备。用户程序要访问设备，依然可以通过/dev 路径中的设备文件进行访问。但除了/dev 之外，许多设备挂载在/sys/class 路径之下。两者的区别在于：

- /dev 是 udev 在运行时创建的实际设备；
- /sys/class 是内核在运行时导出的设备类，以分类的形式通过 sysfs 来反映硬件连接的层级。

但究竟是用/dev 还是/sys/class，最终取决于个人喜好。实际上，/sys/class 大部分是指向/sys/devices 下属设备的符号链接。不过，读者如果在/dev 下找不到对应文件（比如 gpio），则必须在/sys/class 中寻找。

5.3.3.4　kernfs

Linux 内核从 3.1.4 版开始，从 sysfs 中剥离了创建虚拟文件系统所需的一部分逻辑而独立形成 kernfs。其目的是让其他内核子系统也可以创建自己的虚拟文件系统。请不要与 UNIX BSD 系统的 kernfs 混淆。

5.3.3.5　Linux 版本信息

以 Ubuntu 为例，在命令行中可以查询出 Ubuntu 版本号和 Linux 内核版本号。

```
allankliu@ubuntu-server-vm:~$ cat /proc/version
Linux version 3.13.0-32-generic (buildd@toyol) (gcc version 4.6.3 (Ubuntu/Linaro 4.6.
3-1ubuntu5) ) #57~precise1-Ubuntu SMP Tue Jul 15 03:50:54 UTC 2014

allankliu@ubuntu-server-vm:~$ cat /etc/issue
Ubuntu 12.04.5 LTS \n \l

allankliu@ubuntu-server-vm:~$ uname -a
Linux ubuntu-server-vm 3.13.0-32-generic #57~precise1-Ubuntu SMP Tue Jul 15 03:50:54
 UTC 2014 i686 i686 i386 GNU/Linux
```

此外，还有其他指令可以获得更详细的版本信息。读者可以自行在树莓派和其他嵌入式 Linux 中使用以上三种命令查看 Linux 版本信息。

基于 Linux 的开发越发标准化，采用 Ubuntu Core/Debian 进行定制的方式的比例会逐渐超过从源码编译的方式。所以，采用较新 Linux 内核的概率越来越大。嵌入式 Linux 基本上也已经以 sysfs/kernfs 为主。

5.3.4　文件 I/O 操作

文件 I/O 的最基本方法是 open/close/read/write。这些基本 I/O 函数的共同特点是通过文件描述符或者文件句柄来完成 I/O 操作。文件 I/O 是操作系统提供的功能，Python 依赖于具体操作系统，并提供对应的封装模块。依托这些标准化文件 I/O 操作，许多语言，包括 C/C++/Java/Python/Perl/Lua，甚至 Bash shell 都可以访问底层设备和硬件。

1．open

进行文件 I/O 操作时，要先打开（或创建）对应文件，返回文件描述符，并通过引用文件描述符来操作文件。

2．close

在文件 I/O 操作后，需要关闭文件。系统中会保留一个文件描述符的引用计数器，以计算该文件被多少进程所引用。只有该计数器清零后，系统才会彻底回收所有资源。

3．read

从打开的文件中读取数据，可调用本函数。

4．write

把数据写入文件，可调用本函数。

5．fsync

使用 write 函数返回后，数据提交到系统缓存。此时数据不一定写入磁盘。要确保在最短时间内写入磁盘，正确的做法是使用 fsync 进行系统数据同步。还有一个系统调用函数 sync 是针对整个系统数据进行同步。如果有大量数据修改，系统写盘时间是无法预测的，所以使用 sync

要谨慎，要对数据规模进行控制。

6．lseek

Linux 文件分为顺序访问文件和随机访问文件。可随机读/写的普通磁盘文件支持使用 lseek 进行定位，而管道、套接字文件或 FIFO 则不可以。

7．ioctl

无法归类的操作被归类于 ioctl 函数中，它是文件 I/O 的杂项函数。许多设备文件通过 ioctl 函数封装了设备特有的操作，比如修改寄存器的数值或开关 LED 等操作。设备除了数据通道采用 read/write 操作之外，大多数设备参数配置都需要通过 ioctl 系统调用来实现。

以上函数都来自 GCC 函数库。在 shell 脚本中，对于 ioctl 函数缺乏直接指令支持，需要单独构建一个二进制可执行程序。而 Python 和操作系统的标准库（如 sys）基本上都涵盖了这些需求，在 fcntl 标准库中包括了 ioctl 系统调用。注意，这仅在 Linux 系统有效。

5.3.5 Linux 硬件编程

硬件设备挂接到 devfs/sysfs/kernfs 之前，需要为硬件设备和特定操作系统安装驱动程序。驱动程序采用 C/C++编写。挂接到设备文件系统后，Python 等应用程序才可以通过设备文件系统与设备沟通。

所以，在特定 SoC 上运行 Linux，需要得到半导体供应商提供的 BSP（Board Support Package），其中包括了 SoC 所有片内外设的 Linux 驱动源码。嵌入式系统五花八门，有些片内外设和外接外设接口往往没有统一的编程规范（比如 GPIO/I2C/SPI/ADC/PWM/CAN 等）。不同厂家提供的 BSP 源码实现有所区别，同时还与 Linux 设备文件变化有关。我们需要参考用户指南进行设计。

一般来说，半导体供应商或者 IDH 提供的 Linux 单板机会提供最基础的标准化接口：串口/USB/以太网，这一点与 PC 类似。至于通过 USB/UART 外接的设备，如 GPRS MODEM 或者 USB 蓝牙等，需要联络设备供应商获取 Linux 源码。至于 GPIO/I2C/SPI/ADC/PWM/CAN 等，即使供应商不提供源码，也通常提供挂接到文件系统中的方法。这样应用层开发者就可以通过标准接口去配置和访问。

以下我们通过 Bash shell 来访问 Linux 硬件设备文件。在串口通信之前，需要先使用 stty 工具对于串口波特率、流控等方面进行配置。

```
stty -ixon 115200
```

代表将当前串口关闭流控，波特率为 115200bps。

```
cat /dev/ttyUSB0
```

即读取 ttyUSB0，也就是 USB 虚拟串口发来的信息。

以上是采用 Bash shell 来访问并读取设备文件的例子。但是 Bash shell 毕竟功能有限，所以

可以利用 Python 来实现同样的目的。Python 运行 shell 命令可以有三种方式：

```
os.system('cat /dev/ttyUSB0')

output = os.popen('cat /dev/ttyUSB0')
print output.read()

(status, output) = commands.getstatusoutput('cat /dev/ttyUSB0')
print status, output

subprocess.call("cat /dev/ttyUSB0", shell=True)
```

最后一种是 Python 3 力推的一种方式。本质上这是 shell 脚本编程，只不过在 Python 中调用 shell 命令。除了 shell 命令，Python 还可以利用 sys 模块来对设备进行编程。

blink_led.py：

```
import sys
import time

print "Blinking User LED"
print "Enter CTRL+C to exit"

def ledon():
   led = open("/sys/class/leds/ds2/brightness","w")
   led.write(str(1))
   led.close()

def ledoff():
   led = open("/sys/class/leds/ds2/brightness","w")
   led.write(str(0))
   led.close()

while True:
   ledon()
   time.sleep(.5)
   ledoff()
   time.sleep(.5)
```

在 Linux 中，只要驱动程序已经被挂接在设备文件系统中，利用 Linux“设备即文件”的特性，Python 就可以利用最基础的标准库来访问硬件设备。

5.4　并行接口

计算机系统中常见的各类并行接口，包括 ECP/EPP、硬盘 IDE/PATA、扩展卡用的 ISA、PCI、AGP、CF 等。这些并行接口的目的如下：

- 提供较高的数据吞吐速率；
- 提供海量存储；
- 提供网络通信；
- 提供数据采集等专用目的扩展；
- 实现与其他接口的桥接。

5.4.1 老旧的 PC 并行接口

在老式 PC 中，并行接口（简称并口）速率比 RS232 要快，当时用于与打印机和其他“高速”数据采集设备通信。并行接口有三个标准。

- 标准并行接口：SPP（Standard Parallel Port），支持 4/8/半 8 位传输，4 位速率可达 40KB/s~48KB/s，8 位速率为 80KB/s~150KB/s，即 320kbps~1.2Mbps。
- 增强并行接口：EPP（Enhanced Parallel Port），速率为 300KB/s，即 2.4Mbps。
- 扩展兼容并行接口：ECP（Extended Capabilities Port），支持 DMA 传输，速率同 EPP。

PC 并行接口曾经是大量仪器仪表，如数据采集卡、通用编程器的标准通信接口，现在均被 USB 总线替代了。

5.4.2 高速总线

在计算机主板上，各类并行接口如 ISA、PCI、PCIe、AGP、ATA 等总线演进得非常快，出现了高速串行化的发展趋势：减少引脚数量，采用高速差分对传输。这样做的目的是减少并行总线的 EMI/EMC 问题，同时提高速率。典型案例如下：从 IDE/PATA 到 SATA/mSATA，从 ISA/AGP/PCIe 到 Mini PCI-E，采用 LPC、SMBus、PMBus 控制外设等。

嵌入式系统的演变虽然没有那么快，但是紧随着 PC 行业进行着技术平移，不仅普及了 USB，还增加了 Mini PCI-E/SDIO 等总线的支持。但出于直接外设控制的考量，GPIO 口依然大量使用。GPIO 以及对应的 PWM/ADC 功能都是以 8/16/32 位宽度端口存在的，我们可以将其归类在并行接口。

5.4.3 GPIO

GPIO（General Purpose Input Output），即通用目的输入/输出口。嵌入式系统几乎都有 GPIO，而且往往都是可编程口，并与第二甚至第三功能进行复用，以适应系统定制扩展的需求。

由于许多嵌入式 MCU/MPU 的数字输入和输出是可编程的，即使采用同样的 ARM 内核，不同半导体供应商也会采用自己的片内资源库和定义。GitHub 上有些工程尝试将树莓派和 BeagleBoard 定义在一起。但这仅仅是这两个平台间进行兼容性的努力而已。Python 目前没有完

整的跨平台 GPIO 模块。

一般来说，GPIO 可编程的寄存器往往对 I/O 方向、I/O 口，以及首要功能、第二功能和第三功能进行配置。

- I/O 方向选择，是指将其定义为输入还是输出。
- I/O 口配置，是指 I/O 口配置为开漏（双向口）、推挽输出，还是高阻输入等。
- 功能选择，是指将引脚编程为 GPIO，还是其他特殊功能。
- I/O 路由，相对比较少见。这主要指片内外设如 I2C/SPI 可以自由定义在任意 GPIO 引脚。这在 Cypress PSoC、Atmel FPGA、NXP LPC 系列以及 FPGA 芯片中出现，大大提升了系统灵活性。

正是因为 GPIO 往往和其他模块混用引脚，所以广义上的 GPIO 包括这些模块和引脚在内。而狭义上的 GPIO 往往指满足普通输入/输出数字开关量的引脚。

在一些平台化的嵌入式平台中，如 Arduino、树莓派、BeagleBoard，往往将这些原本灵活的 GPIO 固定下来，以实现扩展板的生态化。否则，过于灵活的定义将使得许多第三方开发板无法使用。

在 Linux 系统中，每个 GPIO 都是一台设备。Python 可以有多种形式来访问 GPIO。在 MCU 深嵌入式系统中，系统可以将 GPIO 相关寄存器直接暴露给 Python 虚拟机。这一点可以查看 PyMite 和 MicroPython 的相关介绍。

5.4.4　Linux 访问 GPIO

Linx 2.6 以后，在/sys/class/gpio/路径中可以查到 GPIOlib 模块提供的 sysfs 接口。

树莓派/sys/class/gpio/路径下的文件及符号链接，请参见图 5-9。

```
pi@raspberrypi /sys/class/gpio $ ls -al
total 0
drwxrwx---  2 root gpio    0 Jan  1  1970 .
drwxr-xr-x 42 root root    0 Feb 24 18:17 ..
-rwxrwx---  1 root gpio 4096 Jan  1  1970 export
lrwxrwxrwx  1 root gpio    0 Jan  1  1970 gpiochip0 -> ../../devices/virtual/gpio/gpiochip0
-rwxrwx---  1 root gpio 4096 Jan  1  1970 unexport
pi@raspberrypi /sys/class/gpio $
pi@raspberrypi /sys/class/gpio $
```

图 5-9　树莓派中的 gpio 文件夹结构

属性文件有 export/unexport，用于在用户空间导出和关闭对应的 GPIO 设备。还有 gpiochip*N* 的文件夹，对应于 gpio*N*，*N* 指的是对应的引脚号。在每个 gpio*N* 的文件中，有对应的属性文件。

树莓派 gpiochip0 所指向的文件夹内容请参考图 5-10。不同的 Linux 单板机，该文件夹下的内容可能存在一些差异。

```
pi@raspberrypi /sys/class/gpio/gpiochip0 $ ls -al
total 0
drwxrwx--- 3 root gpio    0 Jan  1  1970 .
drwxrwx--- 3 root gpio    0 Jan  1  1970 ..
-rwxrwx--- 1 root gpio 4096 Jan  1  1970 base
-rwxrwx--- 1 root gpio 4096 Jan  1  1970 label
-rwxrwx--- 1 root gpio 4096 Jan  1  1970 ngpio
drwxrwx--- 2 root gpio    0 Jan  1  1970 power
lrwxrwxrwx 1 root gpio    0 Jan  1  1970 subsystem -> ../../../../class/gpio
-rwxrwx--- 1 root gpio 4096 Jan  1  1970 uevent
pi@raspberrypi /sys/class/gpio/gpiochip0 $
```

图 5-10　树莓派中的 gpiochip0 文件夹结构

嵌入式开发板中最常用的用户界面往往不是 LCD，而是 LED 和按键。这都是利用 GPIO 构成的。不过，/sys/class/leds 中有专门的 LED 设备文件。GPIO/LED 两者的区别在于控制权在用户空间还是内核空间。比如 LED 调整亮度，可以在用户空间中进行开关，并利用占空比来调整亮度。但是这种方式在复杂的设计中会占用用户资源，所以交给内核来管理其占空比更合理。

5.4.5　GPIO 的 Python 包

树莓派是来自英国的开源 ARM11 开发板，其核心 MPU 来自 Broadcom 的手机芯片。

采用不同操作系统可以获得不同程度的 Python 支持。其中 Raspbian 是专门针对 ARM11 架构编译的 Debian。随着树莓派升级到四核 Cortex-A7，已经可以使用完整 ARM 版的 Debian/Fedora/Ubuntu 和 Python 的各种实现和版本。

树莓派各版本 GPIO 扩展口示意图如图 5-11 所示。首先，安装树莓派的硬件 GPIO 支持包。

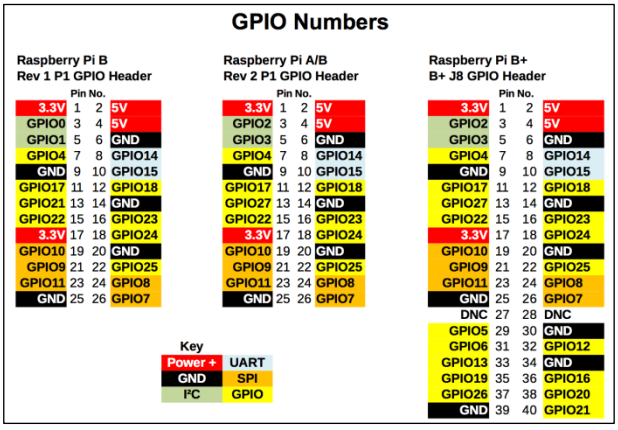


图 5-11　树莓派各版本 GPIO 扩展口示意图

Python 2：

```
sudo apt-get install python-rpi.gpio
```

Python 3：

```
sudo apt-get install python3-rpi.gpio
```

如果安装时出现以下错误：

```
py_gpio.c:23:20: fatal error: Python.h: No Such file or directory
```

说明这个 Python 库需要进行 C 编译，而该 C 文件包含了 Python.h，无法在系统路径中找到。所以我们需要安装 python-dev 包。

```
sudo apt-get install python-dev
```

安装 python-dev 之后，再次安装 python GPIO 支持包即可实现 GPIO 编程。

GPIO 例程 led.py：

```
#!/usr/bin/env python

import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BOARD)
GPIO.setup(11,GPIO.OUT)
while True:
    GPIO.output(11,True)
    time.sleep(1)
    GPIO.output(11,False)
    time.sleep(1)
```

注意　GPIO 操作需要 root 权限。可以使用 sudo 来运行。

```
sudo led.py
```

针对其他 Linux SBC，后面还有针对 BeagleBoard 的 BBIO 扩展包，支持了包括 GPIO 在内的其他外设。

5.5　串行接口

本节描述的串行接口指传统的 UART/RS232 之类的字符型通信接口，而并非 USB 之类的高速串行接口。

5.5.1　异步通信串行口

异步通信串行口指的是一大类通信方式，其特点是基于字符的异步通信，包括我们常见的 RS232/RS422/RS485。

5.5.1.1 RS232

RS232 又被称为 EIA232，即 EIA 组织的推荐标准（RS）。它早期是设备间串行接口，最常见的为 9 针定义（见图 5-12）。其最简单的配置为三线：RXD/TXD/GND，是一种点对点的物理层标准。

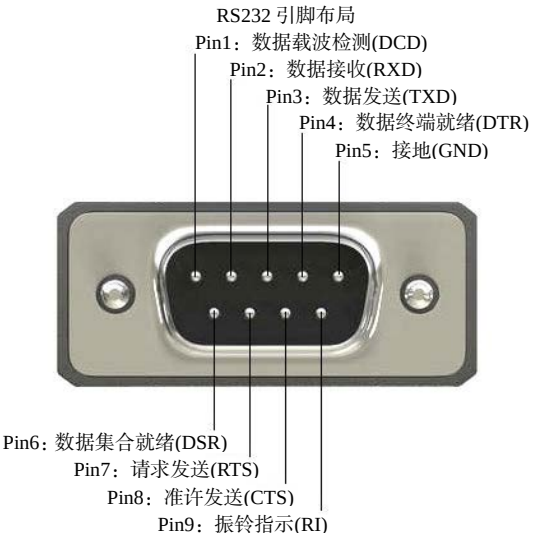


图 5-12 RS232 引脚示意图

在 RS232 开发早期的目的是连接 DTE 和 DCE 通信。其有多种 I/O 定义，其中最主流的 9 针接口如图 5-12 所示。

- DTE，Data Terminal Equipment，用户使用的终端设备，如微机或哑终端；
- DCE，Data Communication Equipment，用于通信的设备，基本上就是各类 MODEM。

RS232 定义之初主要针对电话 MODEM，在 RS232 规范中的某些引脚就是专门用于话务控制的。RI（Ring In）就是有话务拨入的提示信号线。DCD（Data Carrier Detect）用于检测拨号音即载波检测。除了收发 TXD/RXD 之外，其余的许多口线都是对于流量的控制。所有 RS232 都是直通线，但有一种专门用于主机对接的线缆，将主机的 RXD/TXD 对接，由于其中间没有 MODEM，故此被称为 Null MODEM 线。

5.5.1.2 RS422

RS422 由 RS232 发展而来，主要是为了克服 RS232 的传输距离近、速率低等缺陷而设计的。RS422 的标准全称是“平衡电压数字接口电路的电气特性”，它定义了差分平衡通信接口电路。其收、发各有两根线，加上一根地线，共五根线。由于其接收器采用高输入阻抗和发送驱动器，

具备比 RS232 更强的驱动能力，故允许在相同传输线上挂接多个接收节点。

由于所有设备都挂接在同一对双绞线上，所以采用主从模式进行通信以避免冲突。即一个为主设备（Master），其余为从设备（Slave），从设备之间不能通信。所以，RS422 是一种单机发送，多机接收的单向通信。为了拓展应用范围，EIA 又发展了 RS485 总线。

5.5.1.3　RS485

RS485 的电气特征与 RS422 类似。其拓扑逻辑也是主从方式，从设备间依靠主设备进行信息交换。RS485 在工业上的应用要多于 RS422。许多工业标准物理层都基于 RS485，如前面提到的 ProfitBUS、BACnet、CC-Link、DMX512 等。在表 5-14 中对这三种串口进行了总结对比。

表 5-14　串口特性对比表

串口类型	RS232	RS485	RS422
标准引脚数量	DB-9/DB-25	DB-9/25	DB-9/25/37
最小引脚数量	3（TX/RX/GND）	两线制	四线制（YZ/AB）
传输方式	单端/负逻辑电平	差分 TTL 兼容	差分 TTL 兼容
双工模式	全双工	半双工多点主从	全双工
拓扑	点对点	点对多点主从	点对多点主从 256 台设备
最大传输速率	115.2kbps	10Mbps	10Mbps
最大传输距离	20m	10~3000m	10~3000m
缺点	抗干扰性差，传输距离短	抗干扰性强，可组网	抗干扰性强

5.5.1.4　UART/USART

RS232/RS422/RS485 的共同特点是异步字符型传输，MCU 通过片内外设 UART 来支持这些标准。UART 是 Universal Asynchronous Receiver Transmitter 的缩写。伴随着半导体集成度的发展，UART 集成了常见同步通信模式，此类产品被称为 USART，即 Universal Synchronous Asynchronous Receiver Transmitter。所以，现在 UART/USART 可以支持多种串行协议，包括：

- RS232、RS422、RS485，点对点和多机通信，传统异步模式；
- IrDA 模式，可输出同步时钟；
- ISO7816 智能卡模式，可输出同步时钟；
- 单线半双工；
- LIN 总线模式。

通常 UART/USART 通过接口芯片来实现 RS232 设备以及 RS485 总线挂接，而同一设备内的异步通信模块可以直接连接。在某些应用情况下，如 MCU 与数据蜂窝通信模块接口电平不一致时，需要利用三极管或 CMOS 电路做电平转换。这种 UART/USART 配合接口的模式还适

用于 ISO7816/IrDA/LIN 等应用。

由于 UART/USART 的普及，包括 Wi-Fi/BLE/Zigbee/数字蜂窝在内，几乎每种物联网连接方式都有串口模块，通常这些产品被称为“透传”模块或 MODEM。这导致 MCU 集成的 UART 数量急剧上升，市场上出现了集成 5 路 UART 模块的 MCU 品种，连 LPC8XX 引脚数量这么少的品种也可以支持 3 路 UART。

如果 MCU 的 UART 数量无法满足应用需求，可通过定时器中断配合软件来模拟 UART，实现 9600bps 低速率串行通信。

5.5.1.5 USB 虚拟串口

虚拟串口就是没有实际 RS232 物理端口的“串口”，而是在操作系统中以逻辑编程接口替代串口的方式。这包括基于 USB CDC/ACM 类的 USB 虚拟串口，还包括基于蓝牙、红外的串口，以及 TCP/IP 网络串口。

5.5.1.6 pyserial

pyserial 库是一个非常成熟的库，由 Chris Liechti 提供，支持多种操作系统（见图 5-13）：

- Windows，含 Windows XP/Windows Vista/Windows 7/Windows 8/Windows 10；
- Mac OS X；
- Linux，含 x86 版本和 ARM/MIPS 版本；
- BSD，POSIX 兼容 UNIX。

pyserial 除了支持 CPython，还支持 Jython（Java）和 IronPython（.NET/Mono）。该模块会自动选择采用的底层技术。

图 5-13 pyserial 徽标

1．pyserial 的特性

具体特性如下：

- 在所有支持平台中使用同一个类访问串口；
- 通过 Python 属性来访问串口配置；
- 支持不同的字节数量、停止位、极性和流控方式；
- 可以设置接收超时参数；
- 类似文件操作的 API；

- 软件包内采用 100%Python 代码编写；
- 串口设置二进制兼容，不删除 NULL 字节，没有 CR/LF 转换；
- 兼容 Python io 库；
- RFC2217 客户端和服务器端实验性代码，支持套接字桥接虚拟串口。

2．pyserial 的依赖

具体依赖项如下：

- Python 2.2 以后版本；
- 在 Windows 中需要 pywin32；
- 在 Java/Jython 实现中使用，需要使用 Javacomm 组件。

pyserial 经过完整的测试，还有一些帮助类。其文档非常齐全，同时提供了大量的应用实例：

- Miniterm；
- TCP/IP——串口桥接程序；
- 单端口 TCP/IP——串口桥接（RFC2217）；
- 多端口 TCP/IP——串口桥接（RFC2217）；
- wxPython 带 GUI 的应用代码；
- 单元测试。

其内置的 URL Handler 可以支持以下模式：

- rfc2217://<host>:<port>，用于支持 RFC2217 标准的 TCP/IP 串口转发器；
- socket://<host>:<port>，用于非标准 TCP/IP 串口转发器；
- loop://<host>:<port>，用于本地环路测试目的；
- hwgrep://<host>:<port>，用于检索系统内的串口设备；
- spy://<host>:<port>，用于监控目的，可以直接将 RX/TX 线上内容截获。

其中，spy 模式是 pyserial 3.0 之后增加的；而 TCP/IP 串口转发没有内置任何安全算法，所以最好在可靠环境中使用。

3．测试环境构建

为了测试树莓派等开发板对于 pyserial 的支持。笔者构建了以下的运行环境：

（1）LPC812 开发板， 板载 USB CDC/ACM 串口芯片 CP2102；

（2）LPC812 通过串口/CP2102 持续打印开发板版本信息；

（3）在树莓派中运行 Python 代码，引用 pyserial 包，把信息打印在 SSH 终端窗口中。

```
pi@raspberrypi / $ cat /proc/tty/drivers
/dev/tty            /dev/tty        5      0 system:/dev/tty
/dev/console        /dev/console    5      1 system:console
/dev/ptmx           /dev/ptmx       5      2 system
/dev/vc/0           /dev/vc/0       4      0 system:vtmaster
```

```
usbserial           /dev/ttyUSB   188 0-253 serial
rfcomm              /dev/rfcomm   216 0-255 serial
ttyprintk           /dev/ttyprintk   5       3 console
pty_slave           /dev/pts      136 0-1048575 pty:slave
pty_master          /dev/ptm      128 0-1048575 pty:master
unknown             /dev/tty        4 1-63 console
ttyAMA              /dev/ttyAMA   204 64-77 serial
```

将 LPC812 开发板插入树莓派后，看到 CP2102 被识别为 ttyUSB0。

```
pi@raspberrypi / $ dmesg | grep ttyS*
[    0.000000] Kernel command line: dma.dmachans=0x7f35 bcm2708_fb.fbwidth=1136 bcm2
708_fb.fbheight=592 bcm2708.boardrev=0xf bcm2708.serial=0xfc03580d smsc95xx.macaddr
=B8:27:EB:03:58:0D sdhci-bcm2708.emmc_clock_freq=250000000 vc_mem.mem_base=0x1ec000
00 vc_mem.mem_size=0x20000000  dwc_otg.lpm_enable=0 console=ttyAMA0,115200 kgdboc=tt
yAMA0,115200 console=tty1 root=/dev/mmcblk0p8 rootfstype=ext4 elevator=deadline root
wait
[    0.000000] console [tty1] enabled
[    0.529962] dev:f1: ttyAMA0 at MMIO 0x20201000 (irq = 83) is a PL011 rev3
[    0.872183] console [ttyAMA0] enabled
[  463.043591] usb 1-1.3: cp210x converter now attached to ttyUSB0
```

pyserial 代码：

```
#!/usr/bin/env python
import serial

def ser_demo():
    ser = serial.Serial('/dev/ttyUSB0', 115200, timeout=1)
    try:
        ser.open()
    except serial.SerialException, e :
        print "error: %s"%(e)

    while True:
        line = ser.readline()
        print line
    ser.close()

if __name__=="__main__":
    ser_demo()
```

在 TeraTerm 中返回 LPC812 串行输出信息如下：

```
>>>>>>>>>>>>>>>
LPC812-MiniKit.
Firmata under construction.
DevID:00008122
BootloadVer:00000D02
```

```
UID:06056046 AE198903 51A82D22 F5001900
UID (dec):.101015622-2920909059-1369976098-4110424320
A String!
String
This is a 2-digit hex number (char) : 12
This is a 4-digit hex number (short): 1234
This is a 8-digit hex number (long) : 12345678
This is a formatted decimal number: +..24060
```

LPC812-MiniKit 中的代码是笔者采用 ARM mbed 编写的 C++代码，专门用于输出 LPC812 的唯一 ID 号码和其他信息。

这个测试证明：

- 树莓派可以自动识别 USB CDC/ACM 的虚拟串口；
- pyserial 可以工作于树莓派的 ARMv6 版本的 Python 中；
- pyserial 可以工作于虚拟串口。

同时，pyserial 的内置 RFC2217 代码可以将 LPC812 开发板映射到互联网上使用。可以在 GitHub 上下载 rfc2217_server.py 源码进行测试。

```
./rfc2217_server.py -v /dev/ttyUSB0
```

目前 RFC2217 协议没有得到浏览器的支持，所以需要使用 socket 客户端来查看传输的数据。可以使用 Python socket 来打印。更加简单的方式是使用 Linux socket 客户端来访问 RFC2217 端口。

```
sudo apt-get install socket
socket 192.168.1.105 2217
```

socket_client.py：

```
import socket
import binascii

host, port = '192.168.1.105', 2217
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect()
bs = sock.recv(100)
print binascii.hexlify(bs).upper()
```

如果出现乱码，则需要利用工具来查看原因。这些工具包括 RFC2217 自带的日志、Wireshark 抓包软件等，并采用重复发送固定 Pattern 字符串来定位问题所在。pyserial 的 RFC2217 是一个实验性的设计，还不是系统服务，尚需要做一些修改才能真正实用化。

pyserial 代码虽然简陋，但却是最常见的物联网连接方式。通过 USB/UART 连接，甚至可以将 SPI 等高速总线桥接到系统中，构成各种物联网系统。同时通过 pyserial 的 RFC2217 端口

或者其他的 Python 库如 Twisted，直接将串口的码流转发到服务器端。仅需做些应用层协议的适配，即可以构成一个服务器端与设备端之间的数据流逻辑通道。如果配合 RS485/422 的组网能力，则可以快速构建一个简单而完整的物联网应用系统，并一次性涵盖设备组网、联网两个层次。在市场上有类似的产品，检索“串口服务器”就可以找到。

5.5.2 I2C 总线

I2C 是一种可以多芯片并联的中等速率串行总线。I2C 由 Philips Semiconductors 发明，其商标如图 5-14 所示。I2C 总线利用 SDA/SCL 两根线，可以让嵌入式控制器同时对多个元器件进行控制。I2C 总线采用了开漏电路，所以在 SDA/SCL 中可以并联多个 IC，节省了电路板空间，更加重要的是节省了 MCU 的引脚。其速率在 100kbps/400kbps/3.4Mbps 左右。其推出后，很快成为半导体行业的行业标准，尤其在传感器、存储器方面非常流行。

图 5-14 I2C 总线商标

5.5.2.1 I2C 的应用

由于 I2C 的推出时间很早，因此其兼容 IC 种类非常多。在实际应用中，我们也发现过不少与 Philips 标准有出入的 I2C“兼容”集成电路。比如某日本厂家的一枚 RTC，采用 LSB 先出的设计，这导致笔者的一位同事在这枚 IC 上浪费了相当长的时间。使用此类 IC，需要注意阅读 IC 规格，注意与标准 I2C 的区别。

5.5.2.2 I2C 的变种

I2C 支持设备的热插拔，这一特性很适合 PC 外设。ACCESS Bus 是 Philips 和 Intel 将 I2C 用于 PC 行业的一种尝试。笔者记得当时在 Philips 实验室里就有那么一台原型机。插入两个 ACCESS bus 鼠标可以出现两个光标，让两名玩家共同玩一个对抗性游戏。现在看来需要同时出现两个鼠标光标的应用需求很少，不过这在当时的确很新奇。

作为 PC 行业的通用总线，需要诸多厂家从 BIOS、操作系统和应用程序多方面配合。ACCESS bus 没有形成自己的生态，但热插拔和即插即用的思路直接导致了 USB 的研发。Intel 推出的 USB 标准在经过多年的推广后，现在已经成为 PC 的标准配置。

1．SMBus/PMBus

Intel 在 I2C 基础上，定义了 SMBus（System Management Bus）用于系统管理的总线。不能够简单地说 SMBus 就是 I2C，但两者间的确存在继承关系。SMBus 上增加了超时管理和数据传输格式标准，但没有定义传输数据内容。具体如下：

- 数据包错误检测（PEC，Packet Error Checking）；
- 传输超时；
- 标准化传输类型；
- ALERT 报警线；
- SUSPEND 挂起线；
- 电源开关；
- 最大速率为 100kHz/s。

由于 SMBus 和 I2C 的关系，因此 Python 的 I2C 包的名称是 python-smbus。

在 SMBus 基础上，产生了 PMBus（Power Management Bus）。该标准定义了电源控制管理所需的命令和数据结构，以满足控制智能电池组的需求。SMBus 和 PMBus 都是 SMIF 的注册商标。

2．TWI

Atmel 的 TWI，即 Two Wire Interface；其实 TWI 就是 I2C 总线。之所以出现 TWI，是 Atmel 以及其他组织机构为了避免与 I2C 商标冲突而注册的商标。其总线逻辑与 I2C 是一模一样的。

3．显示器 EDID

在显示器行业里，VESA、DVI、HDMI、DisplayPort 中有一个 EDID 标准，该标准将显示参数保存在一个地址为 0x50 的 I2C EEPROM 中。这样显卡可以读取显示设备的显示能力，并相应地进行解析度切换。

4．MCU 主从控制

在电视机领域，一度流行过 Teletext 图文电视。中国也有过自己的图文电视标准 CCST。在最初的两个方案设计中，从控 MCU 专门负责图文电视解码与屏幕显示；而主控 MCU 负责电视机的日常任务，并通过 I2C 总线控制图文电视 MCU。基于 I2C 总线标准，Philips 定义了 SAFARI 协议。该协议可以作为通用控制协议实现多枚 MCU 之间的 I2C 连接。

5.5.2.3　常用的 I2C 器件

I2C 总线发展多年，已经成为 MCU 的标准配置之一，现存的元器件种类非常多：

- I/O 扩展（电平转换，LED 闪烁，调光器）；
- 实时时钟；
- 存储器（EEPROM/FeRAM）；

- 显示器件（LCD/LED）；
- 电容触摸；
- ADC/DAC；
- 传感器（温度、湿度、压强、3D 加速度、空气质量、陀螺仪和磁场强度）；
- 可编程 PMU/LDO/SMPS；
- RFID/加解密芯片；
- PCBA ID 识别等。

5.5.2.4 I2C 应用要点

I2C 在多方面都有应用，无论是硬件还是软件都积累了许多知识点，这里列举一些系统设计方面需要考虑的地方。

- I2C 总线的速度取决于总线上速度最慢的从控 IC，而不是最快的 IC，也就是说最慢的从控 IC 将拖慢整体表现。
- I2C 主控可以采用带硬件 I2C 器件的 MCU，也可以使用一般 MCU 加上两个开漏的 GPIO 来模拟总线时序，只不过模拟速度不如硬件快。
- I2C 主控采用 MCU+GPIO 模拟所需的驱动程序反而少于硬件接口的驱动程序，不过现在这一点资源占用不算特别大的问题。
- I2C 从控设备一般不使用 MCU+GPIO+固件模拟方式。
- I2C 总线上可以有多个主控，形成多主控模式。在多主控竞争模式中，一般不采用 MCU+GPIO+固件模拟方式。
- I2C 总线比 SPI 总线要节省 I/O，可以一拖多。在中低速度情况下，尽量使用 I2C 总线器件。
- 在 I/O 及其受限的模式下，I2C/SPI 可以共享数据线和时钟线，不过固件开发复杂。
- 虽然 I2C 有各种扩展模式、高速模式、长距离模式，但最佳应用还是在同一 PCB 中采用一主多从拓扑和普通运行模式。
- 使用 I2C 兼容芯片时，注意这些器件不兼容的部分，如：位顺序、器件地址和子地址等。
- 由于越来越多的 I2C 器件从地址不再从 Philips/NXP 申请，因此设计 I2C 总线应用时，请注意器件从地址之间的冲突。
- 采用 I2C 总线，不仅仅可以在 MCU 与 I2C 芯片间，还可以在 MCU 之间构成主从结构，减少 I/O 引脚和系统复杂度。

5.5.2.5 树莓派启用 I2C 总线

树莓派中就有多种 I2C 总线的 Python 扩展包，不过仔细阅读过源码之后，就会得出结论：

殊途同归，都是利用设备文件系统来构建的。树莓派默认没有启用 I2C 总线。要想在设备文件系统中找到 I2C，必须在树莓派中启用 I2C 总线。

旧版本 Raspbian 在配置文件中启用 I2C：

```
sudo nano /etc/modprobe.d/raspi-blacklist.conf
```

将禁止 I2C 总线的语句注释掉：

```
#blacklist i2c-bcm2708
```

新版本的 Raspbian 相对简单。具体步骤如下。

第一步，运行 raspi-config 工具：

```
sudo raspi-config
```

第二步，选择 I2C 选项。

如图 5-15 所示，选中“Advanced Options | I2C”，之后选择“Yes | OK | Finish”。重新启动树莓派，I2C 总线就启用了。

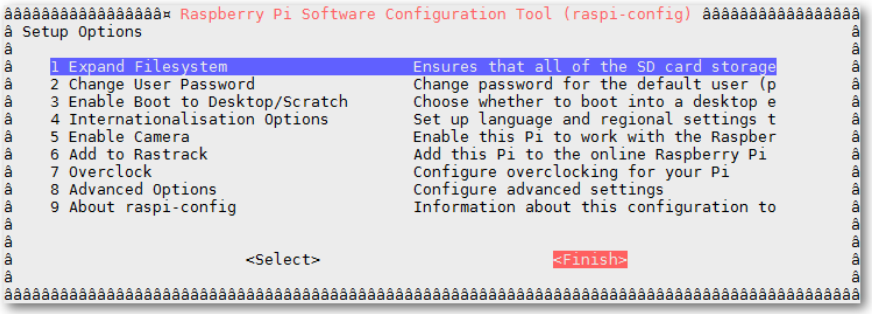

Raspberry Pi Software Configuration Tool (raspi-config)
Setup Options
1 Expand Filesystem
Ensures that all of the SD card storage
2 Change User Password
Change password for the default user (p
3 Enable Boot to Desktop/Scratch
Choose whether to boot into a desktop e
4 Internationalisation Options
Set up language and regional settings t
5 Enable Camera
Enable this Pi to work with the Raspber
6 Add to Rastrack
Add this Pi to the online Raspberry Pi
7 Overclock
Configure overclocking for your Pi
8 Advanced Options
Configure advanced settings
9 About raspi-config
Information about this configuration to
<Select>
<Finish>

图 5-15　树莓派软件配置工具截图

第三步，安装 I2C 工具集。

```
sudo apt-get install -y python-smbus i2c-tools libi2c-dev
```

第四步，查看 I2C 的激活状态。

```
# lsmod | grep "i2c_"
# sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

其他的 Linux 单片机需要对应工具或者安装驱动程序来启用 I2C 总线。

5.5.2.6 I2C 的 Python 包

由于不同架构的 MCU/SoC/CPU 的底层技术有所不同，因此目前还没有一个统一的 I2C 包可供使用，只能够找到对应平台的 Python 包。本节主要介绍依赖于 Linux 系统的 I2C 包，MCU Python 虚拟机的版本在第 6 章中介绍。

1．smbus-cffi

smbus-cffi 包允许访问 Linux 主机/dev 路径下的 I2C 接口。挂载点为/dev/i2c-n。它要求对应的主机内核必须包含 I2C 支持、I2C 器件接口支持和总线驱动。

smbus-cffi Python 包是 python-smbus C 扩展的 cffi 实现，可以工作于 PyPy 和 CPython 2.7。这证实 cffi 是一个不错的跨越不同 Python 实现的扩展方式。而且利用 cffi 扩展和 PyPy 加速，可以适用于一些性能敏感型的应用。

```
from smbus import SMBus
bus = SMBus(4)
bus.write_quick()
reg = 123
bus.write_i2c_block_data(4, reg, [1,4,7])
```

2．python-smbus 安装

请使用 apt-get 来安装 python-smbus。

```
pip install python-smbus
```

3．python-smbus 源码分析

下载 python-smbus 的源码，包含四个文件，其中最主要的就是 smbus.py。该模块基于 Python cffi 构建，i2c 的内核驱动必须挂载在/dev/i2c 中。

```
import os

from .util import validate
from .util import int2byte
from fcntl import ioctl

from ._smbus_cffi import ffi
from ._smbus_cffi import lib as SMBUS
```

从头部导入模块来看，smbus 模块依赖于 os/ioctl 来实现设备文件的读/写开关以及控制，并依赖于_smbus_cffi 作为底层实现。

```
class SMBus(object):
    def __init__(self, bus=-1):
        if bus >= 0:
            self.open(bus)

    def open(self, bus):
        bus = int(bus)
        path = "/dev/i2c-%d" % (bus,)
        if len(path) >= MAXPATH:
                raise OverflowError("Bus number is invalid.")
        try:
            self._fd = os.open(path, os.O_RDWR, 0)
        except OSError as e:
            raise IOError(e.errno)

    def close(self):
        os.close(self._fd)
        self._fd = -1
        self._addr = -1
        self._pec = 0

    def _set_addr(self, addr):
        if self._addr != addr:
            ioctl(self._fd, SMBUS.I2C_SLAVE, addr)
            self._addr = addr

    @validate(addr=int, cmd=int)
    def read_byte_data(self, addr, cmd):
        self._set_addr(addr)
        res = SMBUS.i2c_smbus_read_byte_data(self._fd, ffi.cast("__u8", cmd))
        if res == -1:
            raise IOError(ffi.errno)
        return res

    @validate(addr=int, cmd=int, val=int)
    def write_byte_data(self, addr, cmd, val):
         self._set_addr(addr)
        if SMBUS.i2c_smbus_write_byte_data(self._fd,
                                           ffi.cast("__u8", cmd),
                                           ffi.cast("__u8", val)) == -1:
            raise IOError(ffi.errno)
```

以上仅仅是代码的一部分，而且其顺序被笔者做过调整。读者可以看到在 smbus.py 中，设备的打开和关闭依赖于 os.open 和 os.close 系统调用。而设置从地址采用的是 fcntl.ioctl()方法进行系统调用。各个读/写方法依赖于底层的_smbus_cffi 模块。

在源码中有一个 smbus_cffi_build.py 文件，是构建 smbuf_cffi 模块的源码。在该文件中，定

义了模块名称：_smbus_cffi，并引用了./include/linux/i2c-dev.h 文件。该文件实际上不仅仅是头文件，还是包括了 smbus 的 C 源码。其中包括与 python-smbus 中各个方法对应的各个函数。

```
static inline __s32 i2c_smbus_access(int file, char read_write, __u8 command, int size,
union i2c_smbus_data *data);
static inline __s32 i2c_smbus_write_quick(int file, __u8 value);
static inline __s32 i2c_smbus_read_byte(int file);
static inline __s32 i2c_smbus_write_byte(int file, __u8 value);
static inline __s32 i2c_smbus_read_byte_data(int file, __u8 command);
static inline __s32 i2c_smbus_write_byte_data(int file, __u8 command, __u8 value);
static inline __s32 i2c_smbus_read_word_data(int file, __u8 command);
static inline __s32 i2c_smbus_write_word_data(int file, __u8 command, __u16 value);
static inline __s32 i2c_smbus_process_call(int file, __u8 command, __u16 value);
static inline __s32 i2c_smbus_read_block_data(int file, __u8 command, __u8 *values)
static inline __s32 i2c_smbus_write_block_data(int file, __u8 command, __u8 length,
const __u8 *values)
```

经过编译后，_smbus_cffi 成为 python-smbus 的核心部分。

python-smbus 应用代码：

```
#!/usr/bin/env python
# encoding: utf-8

import smbus
import time

bus = smbus.SMBus(1)

while True:
    for i in range(4):
        bus.write_byte(0x20, (1<<i))
        time.sleep(0.2)
```

4．Adafruit PureIO

Adafruit 是开源硬件行业的重要企业。PureIO 是 Adafruit 针对 I2C 和 SPI 而推出的纯 Python 包，用于直接替代基于 C 扩展的 smbus/spidev 扩展包。

采用纯 Python 的好处是跨平台。不过，本例中的“跨平台”仅限于跨越不同的 Linux 硬件平台。因为 Windows 中没有 sysfs 之类的虚拟文件系统。

5.5.3 SPI 总线

SPI（Serial Peripheral Interface），即串行外设接口。其最初由 Motorola（摩托罗拉）推出，是一种双向同步串行通信接口。SPI 采用 3/4 线接口，收发独立，使用时钟进行同步传输。

SPI 脱胎于移位寄存器进行串/并转换，实现逻辑要比 I2C 简单。其缺点是不同芯片不能够

完全并联挂接，必须使用 CS 片选信号进行区分。如果需要同时传输，那么不同芯片的 SPI 必须彼此独立。基本上一枚 SPI 芯片对应一根片选线，除了一对一方式，还有菊花链的连接方式来解决片选线不足的问题。

在 I2C 市场占有率这么高的情况下，SPI 依然得到保留的原因除了逻辑更加简单、占用硅片面积更小，最大的原因是速度。SPI 总线与芯片内部总线速率在同一数量级，所以高速外设器件几乎还是 SPI 更多。

SPI 在各类 MCU/SoC 中是主要的高速通信接口。x86 主板中的独立 SPI 设备是 BIOS/UEFI 所用的 SPI 闪存。著名的 CIH 病毒就是以擦除 PC BIOS 来危害系统的。Intel 主板设计中预留了 SPI 接口用于升级 BIOS。除了该接口，x86 主板上几乎没有其余的 SPI 设备，所以没有适用于 PC 的 SPI 包。出于安全考虑，也不推荐直接访问 SPI Flash BIOS。

但是，Python 依然在这个领域有所涉猎。Intel 的 Dr. Josh Triplett 在 PyCon 2015 上演示了在 GRUB 引导程序、BIOS 和 UEFI 中运行的 Python 2.7，其主要目的是测试 PC 初始化环境。该 Python 包名称为 BITS（BIOS Implementation Test Suite），BITS 充分利用了 Python 在自动化测试领域的强项。

5.5.3.1　SPI 的应用

SPI 时钟频率可以与 CPU 内部时钟在同一个数量级上，非常适合高速外设。所以，SPI 是嵌入式和物联网中非常重要的总线技术。SPI 芯片主要集中在以下方面：

- RFIC： IEEE 802.15.4，SubGHz；
- 点阵显示屏：LCD/LED/OLED/EPD；
- 存储器：EEPROM/Flash/MMCSPI Flash；
- 高速 DAC/ADC；
- 传感器、RFID；
- 总线桥接：转接 UART/I2C/USB/Wi-Fi/以太网。

5.5.3.2　SPI 的变种

SPI 也有许多兼容或者类似的总线，其变种如下：

- QSPI，采用队列缓冲器的 SPI 外设，以避免 SPI 数据反复中断 CPU 主循环运行，可简化系统设计。
- Microwire 或 uWire，来自 National Semiconductors，是 SPI 的子集。
- 多数据线 SPI，采用两根或者四根数据线，加大数据传输率。
- mSPI，miniSPI。
- eSPI（enhanced SPI），即 LPC，来自 Intel。

5.5.3.3 SPI 的时序模拟

由于 SPI 时序比较简单，因此设计者可以通过软件操纵 GPIO 模拟 SPI 总线时序，缺点是速度不如硬件 SPI 外设。而且软件操纵 GPIO 模拟 SPI 会占主控的计算时间，其仅适用于 MCU，不太适合 Linux 主机实现。

5.5.3.4 SDIO

SD/MMC/TF 卡是非常普及的接口，其数据 I/O 有四根。SD 卡接口可以兼容 SPI 总线。一般来说，SD 卡基本上都以文件系统接口出现。虽然 SDIO 规格已出现了很久，但实际上进入主流应用市场中的 SDIO 种类很少。最常见的是 Eye-Fi SD 卡。其主要原因有两方面：一方面，目前主流移动设备基本不再提供标准 SD 卡插座；另一方面，SDIO 针对的移动设备接口已经无线化，被 BLE/Wi-Fi 替代了。

有些开发板以 SDIO 接口的 Wi-Fi/BLE 模块作为板载模块，而非 USB 接口。而这些 SDIO 设备将作为网络设备接入操作系统内核，这一点和 USB 有点儿类似。这些 SDIO 作为存储器和网络设备，需要通过文件系统和网络端口进行访问，而不是通过设备文件进行访问。

5.5.3.5 Linux spidev

Linux 系统中的 SPI 驱动名为 spidev（SPI device）。这个驱动中主要是 SPI master 驱动。

许多芯片采用 SPI，是因为需要一个简单而高速的接口来做传感器的数据采集或网络传输，而基于硬件的实时数据处理并非 Python 的强项。笔者认为比较好的模式是将 SPI 设备使用 C/C++封装成抽象设备，将 API 留给 Python 进行访问。应避免频繁地让 Python 访问底层 SPI 总线。例如，WSN RFIC 多采用 SPI 总线，如果 Python 应用程序通过 SPI 总线访问这些设备寄存器，虽然迭代速度快，但是操作频繁，速度也慢。换一种方式，将 RFIC SPI 驱动改写为更加抽象的设备，比如网络设备或者基于寄存器的 API 函数，通过 ioctrl 提供，这样或许更加快捷。

5.5.3.6 树莓派启用 SPI 总线

在树莓派中，SPI 总线与 I2C 总线的启用步骤类似。

旧版本 Raspbian 在配置文件中启用 SPI 总线：

```
sudo nano /etc/modprobe.d/raspi-blacklist.conf
```

将禁止 I2C 总线的语句注释掉：

```
#blacklist spi-bcm2708
```

新版本的 Raspbian 相对简单。具体步骤如下。

第一步，运行 raspi-config 工具：

```
sudo raspi-config
```

第二步，选择 SPI 选项。

如图 5-16 所示，选中“Advanced Options | SPI”，选择“Yes | OK | Finish”。重新启动树莓派，SPI 总线就激活了。实际上，除了 raspi-config，还有在 Raspbian 中的 Raspberry Pi Configuration 工具，或者直接修改/boot/config.txt 也可以实现配置 SPI 总线的目的。

第三步，查看 SPI 的激活状态：

```
# lsmod | grep "spi_"
```

其他的 Linux 单片机需要查询对应的工具来启用 SPI 总线。

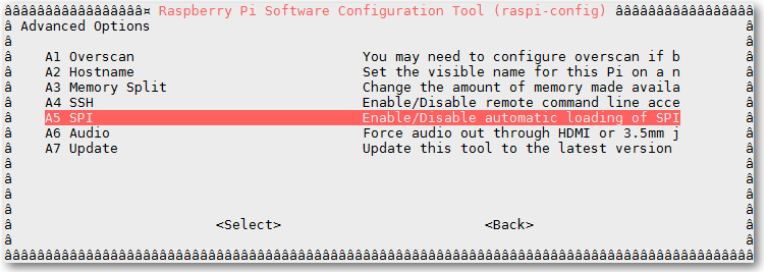

Raspberry Pi Software Configuration Tool (raspi-config)
Advanced Options
A1 Overscan
You may need to configure overscan if b
A2 Hostname
Set the visible name for this Pi on a n
A3 Memory Split
Change the amount of memory made availa
A4 SSH
Enable/Disable remote command line acce
A5 SPI
Enable/Disable automatic loading of SPI
A6 Audio
Force audio out through HDMI or 3.5mm j
A7 Update
Update this tool to the latest version
<Select>
<Back>

图 5-16　树莓派软件配置工具启用 SPI 截图

5.5.3.7　Python spidev 包

Python 官网上有对应的 spidev 包，可以采用 pip 安装。网址如下：https://pypi.python.org/pypi/spidev/3.1

python-spidev 代码：

```
#!/usr/bin/env python
# encoding: utf-8

import spidev
import time

def demo():
    bus = 0
    device = 0
    spi0 = spidev.SpiDev(bus, device)

    spi0.writebytes(range(10))
    x = spi0.readbytes(10)
    print x

if __name__=="__main__":
    demo()
```

5.5.3.8 其他 Python SPI 包

除了官网 spidev，在 GitHub 上还找到了多个 spidev 相关的 Python 包，但其衍生的分支数量都不高。

1．SPI-Py

作者：Louis Thiery，源码名称：spi.c，它是基于 CPython 的 C 扩展，兼容 Python 3.X。其中有个测试例程 test-nRF.py，是利用该模块构建的测试程序，用于通过 SPI 与 nRF24L01 通信。

2．py-spidev

作者：Volker Thoms（德国），源码名称：spidev_module.c，它是一个适用于 Linux CPython 的 C 扩展。和 cffi 不一样的是，其不适用于 PyPy。

3．python-spi

作者：Tom Stokes，这是一个基于 spidev 的纯粹 Python 包。其源码都是基于 ctypes 和 fcntl.ioctl 系统调用而构建的。

4．Adafruit Python PureIO

其还处于开发阶段。GitHub 上仅有 smbus.py 包。

读者可以自行下载这四个软件包，选择其中一种用于自己的工程。因为物联网的关系，推荐阅读一下 SPI-Py 的 nRF24L01 源码，该工程可以快速拓展到其他 RFIC。

GPIO+I2C+SPI+UART 基本上可以覆盖大多数数字电路和模拟电路。即便没有 ADC/DAC 等模电电路，也有替代方案：DAC 可以用 GPIO 配合定时器实现 PWM，继而构成 DAC 电路；定时器配合外部电压比较器和 GPIO 开关可以构成 Sigma-Delta 的 ADC 电路。如果需要高精度、高速率、不同的供电电压的模拟电路，可以使用 SPI 或者 I2C 总线连接外部 ADC/DAC 电路。

由于运行在树莓派中的大多数不是实时操作系统，在外部事件发生时，操作系统上下文切换消耗较大，因此无法实施响应外部事件，操纵 GPIO 的定时精度不是很高。如果增加 MCU 来分担这些实时响应任务，就可以实现外部事件的实时响应，以及精确的 GPIO 控制，MCU/MPU 间可以通过 UART/SPI/I2C/USB 来沟通。实际上，相当多的消费产品中均有此设计，比如 NXP 的机顶盒 SoC 中就采用了 MIPS+8051 的结构，8051 负责 Bootloader 和红外解码。国内某所大学曾经设计过一款树莓派扩展板，采用 STM32F103 为树莓派提供实时特性支持和模拟电路支持。

5.5.4 与其他硬件平台相关的 Python 包

前面以树莓派为例介绍了异步串行口、I2C、SPI 的使用，其他的嵌入式平台也有专用 Python 包。

5.5.4.1 BeagleBoard

与树莓派类似，BeagleBoard 是另外一种流行的嵌入式单板机，两者有很多类似的地方。单

从 Python 编程驱动硬件角度来看，两者使用了不同的 Python 包。看来跨平台的 Python 硬件包还有待时日。

1．BBIO

Adafruit 公司为 BeagleBoard 提供了一个开源的 Python 包：BBIO。它需要 Linux Kernel 高于 3.8。除了 BBIO，Adafruit 还有一个 PureIO，但不成熟。

BBIO 包括以下硬件相关的类：

- GPIO；
- PWM；
- ADC；
- I2C；
- SPI；
- UART，也依赖于 pyserial 包。

2．安装 BBIO

登录 BBB 之后，输入：

```
pip install Adafrui_BBIO
```

也可以从 Git 中抓取最新的代码：

```
git clone git://github.com/adafruit/adafruit-beaglebone-io-python.git
#set the date and time
/usr/bin/ntpdate -b -s -u pool.ntp.org
#install dependency
opkg update && opkg install python-distutils python-smbus
cd adafruit-beaglebone-io-python
python setup.py install
```

示例如下：

```
#!/usr/bin/env python
import Adafruit_BBIO.GPIO as GPIO
import Adafruit_BBIO.PWM as PWM
import Adafruit_BBIO.ADC as ADC
import Adafruit_BBIO.SPI as SPI
import Adafruit_BBIO.UART as UART
import serial

import time

UART.setup("UART1")
GPIO.setup("P8_12", GPIO.OUT)
GPIO.output("P8_12", GPIO.HIGH)
GPIO.cleanup()
```

```
ADC.setup()
value = ADC.read("P9_40")
print value

#PWM.start(channel, duty, freq=2000, polarity=0)
PWM.start("P9_14", 50)
PWM.start("P9_14", 2, 10, 0)
```

可以参考 Adafruit 的例子，也可以查看 GitHub 上某位北京的工程师所提供的基于 BBIO 的测试代码。

5.5.4.2 MRAA

MRAA（读作 em-rah）来自 Intel，它是用 C 语言编写的低级别外设通用库。MRAA 旨在提取与平台（比如英特尔 Galileo 或英特尔 Edison 开发板）基础 I/O 功能访问和控制相关的详细信息，并将其转化为简洁的 API 提供给高层应用程序使用。

MRAA 可作为 Linux GPIO 设备之上的转换层。尽管 Linux 提供了丰富的基础设备文件来控制 GPIO，其用于处理 GPIO 的通用指令也非常标准，但其使用难度相对较大。

不同的硬件平台之间差异明显，拥有不同的功能、引脚编号和 GPIO 类型。更换平台后，同一编号的 GPIO 引脚可能无法支持相同类型的功能。某个特定平台可能根本没有这个编号的引脚。另外，GPIO 在平台上的配置方式也取决于不同因素。例如，一种 GPIO 引脚使用模式可能会影响其他引脚的其他使用模式，或影响其他引脚的使用。MRAA 可降低程序开发的复杂度。尽管 MRAA 可用于编写独立于平台的代码，但开发人员仍然需要确保代码足够耐用，以适应平台的各种局限性。

MRAA 支持 Java、Python、JavaScript 对于底层硬件的调用，支持 GPIO、ADC、PWM、外部中断、I2C、SPI 和 UART 的驱动。MRAA 使用 CMake 作为构建工具。

Intel 为自己的硬件创建了一个多合一的软件库，实现“跨平台”目标，但是使用与其他开源项目不一样的构建工具和实施手段。Intel 的目的很明确，通过所谓跨平台，吸引开发者从 ARM 平台过渡到 Intel Quark 平台。

MRAA 设计隐含的意义如下：硬件外设需要一个统一的 API，以独立于平台的方式提供给高层的应用程序。不过我们需要观察，谁最后会“统一”API。

就此，我们针对基础 I/O 的介绍结束了。接下来我们要介绍：USB。

5.6 USB 总线

USB 是计算机的行业标准之一，尤其是 Windows 95 之后，其成为许多外部设备的标准接

口。这也迫使许多老式接口如打印机并口退出了市场。许多必须保留的接口如 RS232，则通过 USB/RS232 的方式保存。中国规定手机充电口必须采用 USB 标准后，许多手机厂家的私有 PC 接口也都被统一为 USB。

USB 不仅仅替代了老式低速端口，而且还嵌入了其他的总线接口。PCI Express Mini Card，缩写为 Mini PCI-E，就内置 USB 支持。仔细观察 3G/4G MODEM 模块，Mini PCI-E 其实都是利用了内置 USB 接口和主机通信。所以，零配件市场出现了 USB/Mini PCI-E 的转接卡。这种情况在 PC/路由器等 MODEM、Wi-Fi、BLE 等网络设备中特别流行。

5.6.1　USB Endpoints

USB Endpoints，即端点，也就是所谓的逻辑连接点。这可以理解为 USB 的单一功能：U 盘采用 MSD，串口转接器采用 CDC，而复杂一点儿的设备具备多个 Endpoints。如手机在不同模式中采用不同的 Endpoints。从笔者的角度来看，一个基础的物联网模块至少需要两个基础 Endpoints：CDC 和 DFU，前者负责正常通信，后者负责固件更新，而且无须插拔模块即可以完成远程管理。

5.6.2　USB Device/Host/OTG

USB 分为主机（Host）和设备（Device）两种角色。针对移动设备，还有 OTG（On-The-Go）规格。一部手机接入计算机时作为设备，而接入 USB 盘时则作为主机，这两者间的切换不仅涉及 USB ID 引脚，还涉及锂电池升压电路和 VBUS 的供电切换电路。早期相当多的 Android 手机虽然芯片标配 OTG，但却因为手机供应商要节省升压电路和切换电路成本而直接“阉割”了 OTG 功能。最近推出的手机则纷纷将 OTG 作为标准配置，这样许多外设就可以非常容易地接入 Android 设备。

5.6.3　USB 3.0

USB 2.0 的传输速率为 480Mbps（60MB/s）半双工，而 USB 3.0 的传输速率为 5.0Gbps 全双工。由于 USB 3.0 中采用 10 位传输（包括纠错码），所以其全速实际为 500MB/s，这为 USB 2.0 的 8 倍左右。USB 3.0 在物理接口和传输方式上与 USB 2.0 存在着较大差异。

5.6.4　libUSB

驱动开发是内核开发中工作量较重的一部分。了解内核以及购买开发工具对于开发者和开发预算要求都比较高。Linux 平台的 USB 驱动开发，主要有内核开发和基于 libUSB 的无驱动开发两种形态。对于常见的设备如 HID（键盘鼠标）、MSD（U 盘、移动硬盘），系统内核都已经

加载了驱动，开发者无须关心。而某些与应用密切相关的客户设备，采用 libUSB 是一种很不错的方式。毕竟基于内核开发，USB 的复杂度比 UART/SPI/I2C/GPIO 要高许多。此外，libUSB 隐藏了操作系统的底层细节，可以实现跨平台使用。

libUSB 设计了一系列外部 API 以供应用程序调用。这些 API 调用了内核底层接口，和内核驱动程序中所用到的函数相差无几。这种用户态驱动 API 架构使得 libUSB 使用也比开发内核驱动程序要容易。不过由于性能和其他方面的考虑，一些新的或者比较特殊的 USB 设备，比如传输实时音视频的设备，还是需要依靠内核开发模式开发驱动程序。

5.6.5 PyUSB

PyUSB 库是 Python 在用户级的 USB 支持，是 libUSB 的 Python 封装。前面提到，USB 有不同种类的设备对应不同的 Endpoints，除了 USB CDC/ACM 之外，消费者常见的为 USB HID/MSD 等设备。因为 pyserial 已经支持了 USB CDC 设备，而 USB 的 HID/MSD 设备可以通过其他系统调用获得，所以 PyUSB 一般用于支持这些设备之外的种类，比如 DFU、Debugger 和自定义设备。更加重要的是，在 x86 等 PC 平台中，甚至以后的高性能 SoC 中，USB 可能是转接 UART/I2C/SPI 以及其他接口的唯一途径。笔者在网上看到有开发者因为 PyUSB 不支持同步等时（isochronous）模式，而自己重新封装了 libUSB。现在尚不知其封装后的效果如何。

有鉴于 USB 开发的复杂度，PyUSB 可能算是最复杂的硬件相关的 Python 包了。开发之前最好要有思想准备。开发前阅读 USB 文档、芯片文档、libUSB 文档和 PyUSB 文档是必不可少的。

采用 PyUSB 主要适用于 Windows 和完整版 Linux 平台中的主机侧的设备驱动。USB 设备一般都基于资源特别受限的 MCU 之上，如 STM32F103 或 LPC11U24 等，不适用于 PyUSB。设备侧的 USB 大多数基于 C/C++实现。即便是 MicroPython，也都是使用 C/C++ USB 堆栈来完成的。

在 Android 平台上的 USB Host API 类似于 libUSB/PyUSB 的用户空间开发方式，采用 Java 开发。PyUSB 访问 USB 设备也没有得到验证过。因为 PyUSB 是 libUSB 的绑定，不是 Android Java 类的绑定。理论上需要使用另外的 Java/Python 绑定工具：Pyjnius。这需要两次封装过程，笔者也没有看到过实际例子。

PyUSB 的开发步骤如下：

（1）使用现成的 USB 桥接 IC 或者 USB MCU 开发 USB 设备（硬件和固件）；

（2）插入主机后，使用 Linux lsusb 或者 Windows USBview 工具来查看枚举后的 USB Endpoints；

（3）收集目标设备的 VID:PID 信息，这是设备的驱动索引；

（4）根据 VID:PID 和 Endpoints 开发驱动和应用程序。

```
allankliu@ubuntu:~$ lsusb
Bus 001 Device 004: ID 1a86:7523 QinHeng Electronics HL-340 USB-Serial adapter
Bus 001 Device 002: ID 80ee:0021 VirtualBox USB Tablet
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

以上是笔者运行 lsusb 返回的信息：找到 root hub 设备、USB/UART 转换 IC CH340G。

```
>>> import usb
>>> d = usb.core.find()
>>> d
<DEVICE ID 1a86:7523 on Bus 001 Address 004>
>>> d.backend
<usb.backend.libusb1._LibUSB object at 0x7f2ea17ff950>

>>> dir(d)
['_Device__default_timeout', '_Device__get_def_tmo', '_Device__get_timeout', '_Devi
ce__set_def_tmo', '__class__', '__del__', '__delattr__', '__dict__', '__doc__', '__f
ormat__', '__getattribute__', '__getitem__', '__hash__', '__init__', '__iter__', '__
module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__s
izeof__', '__str__', '__subclasshook__', '__weakref__', '_ctx', '_do_finalize_object
', '_finalize_called', '_finalize_object', '_get_full_descriptor_str', '_langids', '
_manufacturer', '_product', '_serial_number', '_str', 'address', 'attach_kernel_driv
er', 'bDescriptorType', 'bDeviceClass', 'bDeviceProtocol', 'bDeviceSubClass', 'bLeng
th', 'bMaxPacketSize0', 'bNumConfigurations', 'backend', 'bcdDevice', 'bcdUSB', 'bus
', 'clear_halt', 'configurations', 'ctrl_transfer', 'default_timeout', 'detach_kerne
l_driver', 'finalize', 'get_active_configuration', 'iManufacturer', 'iProduct', 'iSe
rialNumber', 'idProduct', 'idVendor', 'is_kernel_driver_active', 'langids', 'manufac
turer', 'port_number', 'port_numbers', 'product', 'read', 'reset', 'serial_number',
'set_configuration', 'set_interface_altsetting', 'speed', 'write']
```

这是一个简单的例子。不过千万不要将 root hub 导入 PyUSB，那会导致目标设备被应用程序使用。其他的设备插入 hub 中也可能就不工作了。所以在用户空间使用 libUSB/PyUSB/USB host API，**必须针对特定设备，最好是自定义设备**，否则有可能会存在此类问题。如果 PyUSB 的开发目标是 FTDI 之类的标准桥接 IC，应用程序也需要和目标设备 MCU 交换信息，确认是否是目标设备。因为 FTDI 芯片的市场占有率很高，许多其他设备也会采用 FTDI 芯片开发，所以很容易出现占用其他供应商设备的情况。

5.6.6　标准化 USB 桥接

无论是 PC 还是嵌入式，USB 都是仅次于 UART 的标准化接口。虽然 USB 开发难度相对较大，但是通过 USB 桥接其他接口可以实现系统的标准化开发。

笔者在构思通用物联网网关时，其中最重要的设计重点就是将各种连接 IC 通过 UART/USB 双接口接入主控平台。其中 UART 适用于大多数 MCU，而且适用于低功耗设计。USB 设备可

以采用低端 USB MCU，而 USB 主机可以是 Cortex-M3/M4、Cortex-A MPU/SoC 或标准 x86 平台。

假设 RFIC 的对外接口为 SPI，速率为 2Mbps 左右，那么可以通过两种形式转接到 USB。

（1）USB/SPI 转接 IC。比如 FT220/FT2232 等，其接口直接，无须 MCU，优点是无须固件开发。

（2）USB/UART 桥接+MCU。由于 USB/SPI 芯片比 USB/UART（CP2102/PL2303）贵，所以中间利用 MCU 再做一次桥接。其缺点是增加了 MCU。

（3）USB MCU。比如 STM32/LPC/KL，实现 SPI/USB 和 UART/USB 转换。其优点是通过协议和属性来区别不同的 RFIC，而且转换不同通信方式的代价很低，不需要准备不同的库存；缺点是需要额外的固件和主机驱动开发。

笔者的目标是利用方法 3 实现商品化。利用这种 USB+UART/SPI/I2C 的转换实现通用物联网网关的优缺点如下。

优势

- 标准化封装（UART/USB/Mini PCI-E），可以统一机械外形、电气接口。
- 堆栈可以在主机实现，主机计算资源比 MCU 要多，堆栈不再受限于 MCU。
- 堆栈 IP 化。在主机中实现的堆栈可以进一步通过 TCP/IP 隧道在云服务器中实现，迭代速度非常快。
- 模块热插拔。充分利用 USB 特性可以实现模块热插拔，并即时转换协议栈的物联网网关。
- 容易扩展。USB 比 UART/SPI 的可扩展性强了许多。
- 跨平台实施。模块可以插入手机 OTG、Android 盒子和智能电视、路由器，配合对应的 APP 和软件，可以集成物联网网关的功能。
- 易于固件下载，USB 可以实现 DFU/MSD Bootloader。

缺点

- 增加硬件成本，增加 USB MCU。
- 增加调试难度，增加 USB 堆栈。
- 增加开发成本和时间，主要指最初的概念开发。
- 增加通信延时。没有在 MCU 内部实现堆栈的确增加了网络通信延时。
- 替代方案风险。某些堆栈开发已经或者正在被操作系统标准化，比如 6LowPAN/Zigbee/IEEE 802.15.4 等，存在开发过程中被替代的风险。

这个设计面向的不是标准化网络设备如 BLE/Wi-Fi/Zigbee，而是各类 SubGHz/2.4/5.8GHz IoT 私有 WSN 堆栈以及新的物联网接入技术。不过即使是针对私有协议，也应该提供该模块的套接字服务，这更加符合物联网的本质。

设计将完全采取标准化开发的原则。之前考虑过 STM32F0/F1 MCU，但是发现该系列 MCU

没有将 USB 移植到 mbed C++平台，只能够通过 STM32F0/F1Cube + Keil，选择 LPC11UXX/KL2X + mbed，或者基于 ARM 的 Arduino 进行开发。日后可能考虑采用整合性 SoC 进一步降低成本。

5.6.7　与 USB 相关的其他设计

除了标准化的 USB，某些组织也推出过一些 USB 的衍生设计。

5.6.7.1　USB AOA/ADB for Android

Google 围绕 Android 的 USB 接口展开了一系列设计。首先，它的 APP 调试接口 ADB（Android Debugger Bridge）的物理通道是 USB device。ADB 是基于 USB 的 TCP/IP 端口。所以，现在也可以通过 Wi-Fi 进行调试。之前笔者设计过一款开发板 GAP，就是利用 ADB 调试口实现 Android 与 KL25Z MCU 之间的 USB 通信。但是，Android 4.0 ICS 之后 ADB 默认使用 RSA2048 认证。这对于资源有限的 MCU 来说，计算时间长，实施难度大，于是其被放弃了。

USB AOA（Android Open Accessory）是 Google 的官方外设接口。它与标准 USB 规范存在的最大区别在于：VBUS 的供电方向。在 AOA 模式下，VBUS 由外部设备供电给 Android 主机，但是 Android 却可以控制外部设备。这和 USB Device/Host 相反。这样做的好处在于，Android 主机可以在持续充电/供电情况下控制外设。但是这种模式现在看来并不流行，开发有一定难度。但到目前为止，市场上很少看到有 AOA 设备在出售。在大部分场景中，USB AOA 可以被 Wi-Fi 和 BLE 所替代。一些需要持续工作的场景，可以通过大容量充电宝或交流适配器来解决。

5.6.7.2　USB Host API for Android

Android 的 USB Host API 是 libUSB 在 Android 平台中的 Java 封装。从这个角度来说，不太复杂的 USB 设备都可以被直接接入 Android 设备。其唯一的缺点是：需要 Android 设备对外设进行供电。这对大多数手机来说，续航能力和供电电流将成为应用的瓶颈。

USB Host API 及 BLE/Wi-Fi 是 Android 平板、手机的主要外接方式。

5.7　Linux 网络设备驱动

Linux，或者说 UNIX 主机，天生就是联网的。就连系统内部通信都采用套接字通信方式。

5.7.1　TCP/IP 套接字编程

TCP/IP 的历史非常悠久，各类编程语言都提供 TCP/IP 包。不过嵌入式，尤其是深嵌入式系统支持 TCP/IP/HTTP 等协议是最近 10 年才发生的事情。毕竟在 PIC/8051 此类资源受限的 MCU 上实现还是蛮有挑战的。

越来越多的物理端口逐渐消失，逐渐都抽象化为 TCP/IP 端口。这一点从 Android 调试桥中就可以看得非常清楚。一般嵌入式开发往往采用物理端口，需要 In Circuit Debugger，即在线仿真器。而 Android 却在唯一 USB 接口上，加载 TCP 端口提供 ADB 调试支持。

服务器端的许多系统服务、中间件服务现在也都是以 TCP 端口方式提供的，如 MySQL、Redis、MQTT、TSDB 等。进程间也采用 TCP 端口形式提供，这也是微服务架构的技术基础。

这在许多物理端口受限的情况下，可以视为一种趋势。pyserial 中的 RFC2217 端口转发可以理解为对于传统设备的 IP 化支持。IP 化之后，传统设备联网到中心服务器变得简单了。所以从这个角度来说，即使是私有通信协议，最后也要实现 IP 化，以便使用现有的 IP 工具来管理和规划。这也是 6LowPAN 得以流行的原因。

5.7.2 IEEE 802.3 到 IEEE 802.11

IEEE 802.3 通常指的是以太网，IEEE 802.11 通常指的是 Wi-Fi。从以太网到 Wi-Fi 的普及，体现了物联网从无线到有线的发展趋势。

最早的 MCU 增加 TCP/IP 堆栈是通过串口/GPIO 连接通信模块。比如通过串口连接 GPRS 模块，通过 GPIO 操作 ISA 接口以太网卡，或通过 SPI 连接以太网模块。现在的一些 Cortex-M3/M4MCU 往往内置以太网 MAC，配合外部以太网 PHY 就可以直接上网。在无线通信方面，目前大多数还是两片方案，应用程序运行于单独的 MCU，通过 UART/SPI/SDIO 连接 Wi-Fi/BLE/MODEM 芯片实现 TCP/IP 通信。一些 Wi-Fi/BLE 模块，如 CC3200/nRF51822 等也开始出现应用与 MODEM 集成化的趋势，即采用 self-host 的方式，应用程序和通信堆栈运行在同一 MCU 中。即便原先 I/O 受限的模块如 ESP8266，也可以通过 UART/I2C/SPI 来外挂低成本 MCU 实现 I/O 扩展。

在 Linux 中，通过串口、SPI、SDIO、USB 虚拟串口、PCIe 以及其他方式连接的“网卡”或“MODEM”，都可能被归属为网络设备，并提供套接字 API。当然，被合并到网络设备之前，部分设备可以暂时归类于字符设备。

这些设备驱动程序需要按照网络设备驱动模式开发和挂接，并提供 socket API 给应用程序。对应地，Python 应用程序可以利用 socket 相关扩展包实现网络通信。在这方面，SocketCAN 从字符设备向网络设备的转化提供了一个可以借鉴的实例。

5.7.3 网络通信实现方案

前面在串口开发中已经提到过 pyserial 自带的 RFC2217_server.py，这是一个可以使用的例子。利用 Twisted 做 socket server（套接字服务器），pyserial 做串口监控，可以实现更实用的 RFC2217 Telnet/COM 通信。

此外，Python 自带的 TCP/IP 相关包也是非常丰富的。Python 依托于 Linux，从一开始设计就支持 TCP/IP 通信。其标准库如下：

- socket，底层网络接口；
- ssl，socket 对象的 SSL/TLS 封装；
- asyncore，异步 socket；
- webbrowser，常规 Web 浏览器；
- cgi，CGI（Common Gateway Interface），一种最基本的 Web 服务器；
- cgitb，CGI 脚本 Traceback；
- wsgiref，WSGI 工具和参考实现；
- urllib/urllib2，使用 URL 打开任意网络资源；
- httplib，HTTP 客户端；
- ftplib，FTP 客户端；
- poplib，POP3 客户端；
- imaplib，IMAP 客户端；
- nntplib，NNTP 客户端；
- smtplib，SMTP 客户端；
- smtpd，SMTP 服务器；
- telenetlib，Telent 客户端；
- uuid，UUID（RFC4122）；
- urlparse，将 URL 分解为不同组成部分；
- SocketServer，网络服务器框架；
- BaseHTTPServer，基础 HTTP 服务器；
- SimpleHTTPServer，简单的 HTTP 服务请求处理器；
- CGIHTTPServer，CGI-HTTP 服务请求处理器；
- cookielib，HTTP 客户端 cookie 处理；
- Cookie，HTTP 状态管理；
- xmlrpclib，XML-RPC 客户端；
- SimpleXMLRPCServer，基础 XML-RPC 服务器；
- DocXMLRPCServer，自含文档的 XMLRPC 服务器。

此外，与互联网通信相关联的还有几种库：

- 数据持久类；
- 数据压缩类；
- 文件格式；

- 加密服务；
- 互联网数据处理类；
- 结构化标记语言类。

许多物联网项目最初的需求就是将之前的串口连接方式转换成 TCP/IP 连接。大多数串口通信是点对点通信，而且基本上都采用二进制私有协议。采用 Python 标准 socket 包作为传输层，原有二进制协议就可以满足客户的基本需求了。笔者的第一个套接字服务器就是直接参考 Python socket 文档，配合 struct 辅助包完成的。好简单！

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while True:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()
```

的确很简单，没有几行代码。但是千万不要着急，别忘记这仅仅是一个教程而已。其可以用于实验室和调试，但作为产品投入生产环境却还需要解决很多实际问题，比如：

- 如何维护多个 TCP/IP 连接？
- 如何维护长连接？如何维护短连接？
- 如何记录数据？
- 设备接入时的传输层安全以及设备接入授权？
- 多个连接采用多线程还是多进程？
- 多个连接（进程和线程）之间如何通信？
- 如何避免单个连接堵塞整体表现？
- 以及其他还没有涉及的问题。

笔者也是等这些问题一个个冒出来后才发现，还有许多事情没有考虑周全，直到发现了 Twisted。相比之下，Twisted 作为单线程异步通信框架，解决了大多数问题，其更成熟、更完整。笔者草草看过 Twisted 的文档和书籍后，就果断升级到 Twisted。在此过程中，笔者发现还需要考虑更多其他问题：

- 后台运行 Twisted 所需的守护程序？

- Twisted 的 Web 支持太弱，使用 Klein 还是 Cyclone？
- 是否可以运行在手机？
- 是否可以运行在树莓派中？
- 是否可以运行在 Linux 最小系统中？
- 如何支持 TLS/SSL？
- 如何支持负载均衡？
- 如何支持 MQTT？
- 如何支持 REST API？
- 如何支持客户化 Raw TCP/IP 协议的敏捷开发？
- 如何支持安全沙箱？
- 如何支持收集连接数和应用相关的数据？
- 如何连接 Redis？
- 如何连接 MongoDB？
- 如何连接 Hadoop？
- 如何构建实时配置修改？
- 如何构建消息队列？
- 其他问题。

好在 Twisted 没有让笔者失望。经过最初阶段的纠结和学习，笔者发现 Twisted 有着丰富的社区支持和大量工程可以借鉴。所以，笔者到现在为止还是不断地找轮子，安装轮子，测试系统。

笔者想强调的是，看到代码后别太着急将其投入实用。毕竟这是一个快速变化的环境。Python 中的 TCP/IP 是一个百花齐放的世界。Twisted 技术成熟，历史悠久。现在又开始流行 Lua、Golang、Node.js、swoole、gevent 等各种异步框架或者高并发并行框架和语言，让人眼花缭乱。虽然我们没有足够的时间去学习所有的技术之后再来完成当前的设计任务，但请保持一个开放的学习心态去对待这些技术。

不过，本书依然围绕着 Python 的标准库如 socket 及常见的 Twisted 网络库来展开网络编程，并不打算重新造轮子。当然，读者也可以使用 gevent 配合 socket，与 Twisted 网络库进行性能对比。然后使用 PyPy 加速、Jython 和 IronPython 来测试性能。

5.7.4　私有通信协议栈

无论是采用有线连接，还是无线连接，总会产生大量的通信协议。在 Python 的物联网网关设计中会涉及这方面的开发。通信协议尤其是 WSN 协议的设计已经超越本书的范围。总体来说，WSN 协议（尤其是 MAC）需要平衡如下几个要求。

- 能源效率：包括单个节点的能耗和整体能耗，这一点和节点工作占空比有关。
- 可扩展性。
- 冲突避免机制。
- 信道利用率；如何在空间（指向）、频率（包括带宽）、时间（冲突和时间定标）间妥协。
- 数据延迟。
- 数据流量。
- 节点间公平性。
- 实现复杂度。

各个 RFIC 原厂往往提供一系列简单的 MAC 协议，读者可以好好对比、评估。即便是原厂提供的免费 MAC 设计和源码，也需要用户根据实际情况进行适配。

与传统网络重点考虑节点间的公平性、带宽利用率和实时性不同，WSN 的 MAC 协议主要考虑能源效率，这和 WSN 大多采用电池供电、生命周期较长有密切关联。在百度文库中可以找到一些文档描述各类 MAC 的实现策略和对比，可以满足读者的学术研究需求。MAC 协议需要好好对比，根据应用特性而做出适配。如果读者有兴趣的话，可以使用 NS2 之类的网络模拟程序来测试这些 WSN 的性能和合理性。

传统设计模式如下：通信协议在 MCU/SoC 中实施，使用 C/C++语言开发，而 Python 作为应用层来调用。典型的开发模式如下：将 RFIC+MCU 构成一个私有网络的无线 MODEM；而 Python 通过 USB/UART 接口来接入及管理 MODEM 模块。

如何合理分割实时任务和非实时任务是对系统架构的设计挑战。如果某些空口协议对于延时非常敏感，那么协议最好在实时特性更好的 MCU 中实施。MCU 虽然比 MPU 要简单，甚至没有操作系统，但是越简单的架构，采用中断服务，实时特性反而更好。在消费电子如高清电视、机顶盒等 SoC 中，往往内置一枚 MCU，如 8051 或者 Cortex-M0，甚至是自带状态机的寄存器，道理就是如此。红外遥控总体是一个慢速设备，完整的红外报文长度低于 100ms，但是软件来做位解码需要实时处理，高端 MPU/SoC 切换实时任务代价太高。即使 SoC 的处理能力非常强，也无暇分身处理红外遥控。所以，必须采用硬件或者低端 MCU 来处理红外解码。

前面 IBM LoRaWAN 系统设计中的网络交换机架构给笔者提供了一些全新的设计思路：如果主控处理速度足够高，而且完整报文可以通过 DMA 之类的方式进行传输的话，USB/UART 接口部分的延时在网络协议中可以忽略。这样，直接使用 Python 来访问对应的 USB 或串口，MODEM 将部分 MAC 协议转给 Python 主控来实现也是可以的。

将协议放到 MPU 中，而非 MCU 中的另外一个收益就是 MPU 的资源比 MCU 要多。出于成本和功耗考虑，一般 WSN MCU RAM/ROM 都很小。以 TI 的 CC430RF5137 为例，ROM 只有 32KB，RAM 只有 4KB。其提供的 SimpliciTI 协议仅支持 8 路终端节点的接入。一方面，RAM

约束了终端节点数量；另外一方面，RAM 的限制也约束了高级数据结构的使用。例如，在 MPU 中，使用哈希函数可以很快地查找到对应的节点信息；而采用线性查找，平均返回时间就是节点数量的负相关函数，数量越多，速度越慢。

由 CC430R5137 实现底层 MAC，并将报文转发给主机，由主机实现上层堆栈，这里可以充分利用主机充裕的RAM保存更多节点上下文，并采用哈希函数或者键值数据库保存相关信息。这是笔者规划通用物联网网关的主要原因。从 WSN RFIC、MCU、USB 到 Python 运行环境，累加延时要足够低，才能够实现可以商品化的网关。

5.7.5 短距离无线连接

LR-WPAN，短距离无线连接方式，尤其是 2.4GHz ISM 频段，竞争非常激烈。其在消费场景中逐渐标准化，但是在野外、工业、农业等场景中大多数却是私有化协议。

5.7.5.1 蓝牙与 BLE

蓝牙分为经典蓝牙和蓝牙低功耗。蓝牙 SIG 最近推出了蓝牙 5.0 规格，说明蓝牙 SIG 组织的迭代速度加快，正在不断地紧跟物联网的发展趋势，扩展市场占有率。蓝牙是笔者在各个无线标准中，私有化程度最高的一个类别。在 Windows 领域中，居然大多数蓝牙堆栈都是私有的。微软、CSR、Broadcom 等都是如此。在 Linux 中，主要的蓝牙堆栈是 BlueZ/BlueDroid 堆栈。

Linux 官方的蓝牙堆栈是 BlueZ，是高通公司的开源项目。经过多年的开发，Android 4.1 中 BlueZ 的版本升级为 4.93，它支持蓝牙核心规范 4.0，并实现了绝大部分的 Profile。BlueZ 的最大问题在于软件结构不统一，不同 Profile 的控制方式存在较大差异。

BlueDroid 是 Google 与 Broadcom 联合开发的开源项目。与 BlueZ 相比，BlueDroid 的软件结构比较简洁，从 Android 4.2 之后开始用于 Android 平台。高通公司也开始在自己的参考实现中使用 BlueDroid 替代 BlueZ。BlueDroid 虽然有替代 BlueZ 的趋势，但是某些 Profile 还不完善。BlueZ 目前依然在维护，版本已经升级到了 5.40，还可以继续使用。

Bluez 官网提供的 Profile 列表如下。

BlueZ：

- A2DP 1.3
- AVRCP 1.5
- DI 1.3
- HDP 1.0
- HID 1.0
- PAN 1.0
- SPP 1.1

基于 GATT（LE）Profile：

- PXP 1.0
- HTP 1.0
- HoG 1.0
- TIP 1.0
- CSCP 1.0

基于 OBEX Profile（obexd）：

- FTP 1.1
- OPP 1.1
- PBAP 1.1
- MAP 1.0

5.7.5.2 PyBlueZ

PyBlueZ 是用 C 编写的 Python 扩展，封装了 BlueZ 堆栈，以便让开发者专注于蓝牙的高层应用。

```
import bluetooth

target_name = "My Phone"
target_address = None

nearby_devices = bluetooth.discover_devices()

for bdaddr in nearby_devices:
    if target_name == bluetooth.lookup_name( bdaddr ):
        target_address = bdaddr
        break

if target_address is not None:
    print "found target bluetooth device with address ", target_address
else:
    print "could not find target bluetooth device nearby"
```

5.7.5.3 Zigbee

TI 官网上提供了 Z-Stack Linux 网关设计，其内部结构如图 5-17 所示。该设计的硬件基于 TI BeagleBoard Linux SBC 以及 CC2530/CC2538/CC2630/CC2591/CC2590 USB Dongle。基于 Z-Stack，还支持以下应用层协议：

- Z-Stack-Home；
- Z-Stack-Lighting；

- Z-Stack-Mesh；
- Z-Stack-Energy。

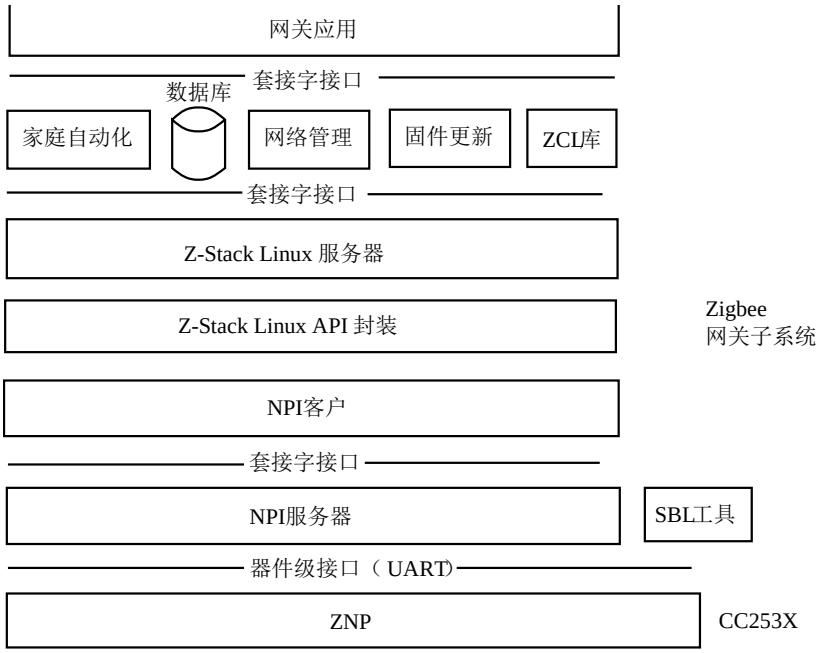


图 5-17　TI Z-Stack Linux 网关示意图

从图 5-17 中可以看出，Z-Stack 对应用层提供的是 socket API。标准化编程，使用什么编程语言就无关紧要了。套接字编程也是物联网的趋势，所有的服务和编程接口都将通过 TCP/IP 进行标准化。

5.7.5.4　6LowPAN

RS-Online（欧时电子）介绍了一款 6LowPAN 路由器，可以将 TI CC2538/CC2560 的 6LowPAN Mesh 网络直接连接到 IBM 的 Bluemix 云端服务。该方案基于 TI 的 Cortex-M3 MCU，并没有使用 Python 进行网络编程，但使用 Python 在主机中对 TI 芯片进行 ISP 编程。

在 Linux 中，Linux-wpan 工程尝试将 6LowPAN 整合进 Linux 内核中。其支持 Atmel/TI/Microchip 的系列 SoC，接口支持 SPI/USB。

6LowPAN 提供的是 IPv6 的 socket 编程接口。对于 Python 来说，这和其他网络编程没有区别。

5.8 工业总线

工业总线太多，其历史渊源和归类很复杂。所以本节仅以 CAN/LIN 两种车用总线为例展开。总的开发方式依然以字符型设备、网络设备为主。前者采用 PyUSB/pyserial/设备文件进行编程，后者采用 socket 进行编程。

5.8.1 CAN 总线

CAN 总线是控制器局域网（Controller Area Network）的简称。它是德国 BOSCH 公司定义开发的串行总线，早期的芯片合作伙伴是 Philips。其最初的开发目的是面向汽车电子应用，减少汽车电子设备的各种线束数量。现在其已成为汽车和工业界常见的标准总线。CAN 总线相关的国际标准为 ISO11898/ISO11519。

和其他总线的不同点是，CAN 总线仅规定物理层和链路层规范。物理层可以是双绞线、同轴电缆以及光纤等。在双绞线 CAN 总线中，即使其中一根线断裂依然可以保持通信。这种高可靠性使得 CAN 总线在工业自动化、农业自动化、船舶军工、医疗设备、工业设备、安全敏感性设备中得到普及使用。

CAN 总线技术优势：

- 网络节点数据通信实时性强，采用抢占式总线仲裁方式；
- 总线拓扑可以为多主多从，形成主机间通信；
- 形成行业标准和国际标准的现场总线。

CAN 总线仅仅规定了 ISO/OSI 的部分物理层和链路层协议，在 CAN 总线之上存在多个高层协议，如：

- SAE J1939，面向大型汽车和重工机械的协议；
- CAN open，由 CiA（CAN in Automation）组织定义的现场总线；
- CAN Aerospace，德国 Stock 航空系统公司定义的航空机载设备通信总线协议，也可以用于无人机；
- DeviceNet，美国 Rockwell 定义的低压开关设备和控制设备控制器设备接口，目前归属于 ODVA 组织，是欧洲和中国标准；
- NMEA2000，简称 NEMA2K 或 N2K，基于 J1939 协议，主要用于船舶导航系统。

1991 年，Philips Semiconductors 制定并发布了 CAN 2.0 规范，可以支持 11 位和 20 位器件地址，并推出了 PeliCAN SJ1000。

在车身电子设计中将 CAN 总线分级，分为高速 CAN 总线和低速 CAN 总线。其中，高速 CAN 总线用于与安全相关的设计。低速 CAN 总线则用于普通节点控制。在一些开源设计如 3DR pixhawk 飞行控制器中， CAN 总线用于智能节点间连接和电调控制，并针对无人机和机器人行

业设计了 UAVCAN 协议和 libuavcan/Pyuvcan 模块。

5.8.1.1　SocketCAN

前面提到在 Linux 中，从 Kernel 2.6.11 开始，Linux 采用了 SocketCAN 的网络设备接口以替代原有的基于字符设备的设计。Linux Kernel 团队给出的理由如下。

大多数现有实现方法仅作为特定硬件的设备驱动，通常基于 Linux 字符设备且功能单一。这些方案通常由字符设备接口来实现原始 CAN 数据帧的发送和接收，直接和控制器硬件打交道。帧队列和 ISO-TP 这样的高层协议必须在用户空间来实现。大多数基于字符设备的实现仅支持一个进程访问。如果更换了 CAN 控制器，那么也要更换设备驱动，且大多数应用程序需要重新调整以适应新驱动的 API。同时字符设备无法复用 Linux 网络队列代码，必须为 CAN 网络重写。

SocketCAN 协议族实现了用户空间的 socket 接口，构建于 Linux 网络层之上，可直接使用队列功能。CAN 控制器的设备驱动作为网络设备注册进 Linux 的网络层。CAN 控制器收到的 CAN 帧可以传输给高层网络协议，要发送的帧报文也会通过高层协议传递给底层 CAN 控制器。传输协议模块可以使用协议族提供的接口注册，所以可以动态加载和卸载多个传输协议。事实上，CAN 核心模块不提供任何协议，也无法在没有加载其他协议的情况下单独使用。同一时间可以在相同或者不同协议上打开多个套接字，可以在相同或者不同 CAN ID 上同时监听和发送（listen/send）数据。几个同时监听具有相同 ID 帧的套接字可以在匹配帧到来后接收到相同的内容。如果一个应用程序希望使用特殊协议（比如 ISO-TP）进行通信，只要在打开套接字的时候选择该协议即可。接下来就可以读取和写入应用数据流了，根本无须关心 CAN-ID 和帧结构信息。

与广为人知的 TCP/IP 协议以及以太网不同，CAN 总线没有类似于以太网的 MAC 层地址，只能用于广播。CAN ID 仅仅用来进行总线的仲裁。

虽然实现 CAN 设备驱动最简单的办法就是直接提供一个字符设备而不使用（或不完全使用）抽象层。但是正确的方法却是要增加抽象层以支持各种功能，如注册一个特定的 CAN-ID，支持多次打开的操作和这些操作之间的 CAN 帧复用，支持 CAN 帧复杂的队列功能，还要提供注册设备驱动的 API。虽然驱动开发复杂些，但应用程序使用 Linux 内核提供的网络框架将会大大简化，这就是 SocketCAN 实现的目的。在 Linux 中实现 CAN 功能最自然和合适的方式就是使用内核网络框架。

从以上描述中可以看到，来自半导体行业的解决方案简单而直接，仅支持少数 IC，缺乏通用性；而来自操作系统内核团队的方案强调通用性和标准化。可以预见到在以后的物联网连接方案中，随着整合力度的加大，类似的过程会不断地发生。

仔细观察 Linux 内核团队的开发网站中关于 networking 这部分的内容。至少以下物联网接入部分已经有部分得到了内核团队的关注：

- IEEE 802.15.4；
- 6LowPAN；
- Zigbee。

总的来说，小众的接入方式，最简单的开发手段是先通过字符设备驱动程序接入系统。比如，各类 SubGHz 的私有 WSN。而某个协议一旦开始普及，就有标准化、网络化编程的可能。

5.8.1.2 Python-CAN

Python 很早已经支持了 CAN 总线编程。随着 Linux 引入 SocketCAN 之后，Python 也转向支持 SocketCAN。Python-CAN 模块支持 SocketCAN、Kvaser CANLIB、CAN/USB 桥接、PCAN BASIC API 等四种接口。高层协议支持 J1939。Python-CAN 支持广播和 Wireshark 抓包调试。

SocketCAN 有两个实现：ctypes 和 Python 原生支持。其中，Python 原生版本需要 Python 3.4 以后的版本。作为测试，可以安装一个虚拟 CAN 卡评估 Python-CAN 软件包。

目前 SocketCAN 无法提供 demo，主要是因为笔者手头缺乏合适的硬件。所以，笔者计划使用 STM32F4 来构建一款 USB/CAN 桥接设备。不过 SocketCAN 也提供模拟设备以方便用户开发高层应用，请读者自行参阅相关文档进行评估。

5.8.2 LIN 总线

局域互联网络（Local Interconnect Network，LIN）针对汽车分布式电子系统而定义了低成本串行通信网络，它是对 CAN 总线和 Flexray 的有效补充。其适用对象是对于网络带宽、性能和容错要求较低的应用：如车门、方向盘、座椅、空调、车灯、湿度传感器、发电机等。

何为有效补充？CAN 总线是扁平化的总线结构，而且采用多主抢占式方式占用总线，这适合一些较高速度的应用；而车灯之类的慢速应用如果占用 CAN 总线带宽则并不合适。所以在 CAN 的节点下设 LIN 二级总线进行分层设计，可以减少这些慢速设备对于高速总线的占用。

LIN 总线基于 UART，而 UART 是所有 MCU 的标配，即使没有 UART 也可以采用 GPIO 模拟。一些文档中提到了 SCI（Serial Communication Interface），这是原摩托罗拉半导体的 68XX 系列 MCU 中的串行口术语，也是 UART 的一种。

LIN 总线的物理层收发器不是简单的 TTL/CMOS I/O，其需要专门的收发器。其物理层采用一根 12V 信号总线和一根无固定时间基准的节点同步时钟线，传输率为 20kbps。LIN 的拓扑多为一主多从结构，由于使用同步时钟，所以总线类型应该归于同步总线。采用 LIN 总线的大多是 MCU 级别控制器。即使在 Linux 和 Python 中，LIN 也应该按照字符设备串口来编程。

5.8.3　其他 ASIC

随着行业整合度的增加，越来越多的接入技术被统一到了少数几种标准化接口，具体接口请参阅表 5-15。

表 5-15　硬件接口与编程接口标准化一览表

硬件接口	编程接口
I2C/SPI/GPIO	smbus/spidev/gpiolib/sysfs
UART	pyserial/socket/sysfs
USB	libUSB/PyUSB
蓝牙	pyserial/PyUSB/PyBlueZ/socket 多种
802.3 以太网	socket
802.11 Wi-Fi	socket
802.15.4	socket/WPAN

光纤、ADSL、Cable MODEM 等用户接入设备也都是以 TCP/IP 形式提供的。

从系统集成角度来看，工业、军事和其他特定行业的 ASIC 如果无法形成产量，则 ASIC 流片成本太高。所以，通过标准化接口连接主控制器是必经之途（也无非是 SPI/USB/socket 这几种选择而已）。即使是整合为 SoC，针对特定平台的外设的驱动程序也可以参考 CAN 总线字符设备/网络设备的开发流程。

5.8.4　定制 Python 扩展

虽然笔者尽了自己的最大努力来收集各种级别的连接性技术，但受限于个人能力，难免挂一漏万。在医疗保健、农业、工业、航空航天和军事领域存在着大量的接口和通信协议没有被写入本书中，而这些行业存在着大量彼此不兼容的现场控制总线。

工业自动化和组态软件应该是物联网的相关应用了。所有自动化设备的遗留总线技术都已通过各类桥接设备最终归并到 IP 网络和标准接口。此外，这些技术的无线化趋势很强，可以通过其特种协议的无线集中器/路由器接入 IP 网络。

前面已经提及了大量的工业总线。实际上无论是工业总线，还是 I2C 和 SPI 总线，都是采用 C API 扩展方式或者 cffi 等其他扩展方式来实现的。只不过，主流接口的扩展已经被开源社区和内核团队实现了，非主流应用的接口扩展则需要开发者自己实施而已。我们可以通过扩展插卡（PCI）、桥接设备、USB 接口、以太网/Wi-Fi 和串口接入系统。

- 扩展插卡，需要使用 Python C/ctypes 扩展 so/DLL 库；
- USB 设备，采用 libUSB + PyUSB 进行编程；

- IP 网络，采用 socket/REST API 进行编程；
- 串口，采用 pyserial 进行编程。

在电子工业出版社翻译出版的《真实世界的 Python 仪器监控》中，详细介绍了如何利用 Windows 的 DLL 文件来开发工业总线应用。虽然这本书围绕着仪表行业进行设计，但其对于日常的 Python 设备相关设计也有着很重要的启示作用。

5.8.5 Windows DLL

Windows 适合商品化开发，这与 Linux 开源生态存在较大差异。除了 libUSB/PyUSB 之外，大多数设备驱动以 DLL 库形式提供。采用 ctypes 包可以无须 Windows 源码即可以调用 DLL 库。

ctypes 包

ctypes 是 Python 的外部函数库（foreign function library），其提供与 C 语言兼容的数据类型，可以非常方便地调用 Windows DLL 或 Linux so 文件中的函数。

```
>>> from ctypes import *
>>> print windll.kernel32
<WinDLL 'kernel32', handle 76f20000 at 22431d0>
>>> print cdll.msvcrt
<CDLL 'msvcrt', handle fd2a0000 at 1dc2e48>
```

5.9 本章小结

本章介绍了物联网的系统组网和联网技术，且逐层分解到主控制器的常见外设接口，并介绍了在深嵌入式和 Linux 操作系统中，这些外设接口的软件接口和编程方式：

- 嵌入式系统的连接能力在不同层次上的多样性是非常丰富的。
- 嵌入式系统与桌面和移动计算一样，呈现无线化、串行化、高速化、IP 化和标准化趋势。
- 外部设备在 MCU/MPU 中都以寄存器的形式存在。
- 外部设备种类实现标准化的途径：GPIO/I2C/SPI/UART/USB/ADC/DAC/PWM/socket。
- Linux 系统中必须通过驱动程序访问外部设备寄存器。
- Linux 系统中采用设备文件方式访问驱动程序抽象化后的“设备”。
- 深嵌入式 Python 设备采用 C/C++语言直接访问寄存器，并使用 Python 封装。
- 外部设备的核心编程语言是 C/C++，Python 与硬件的接口需要依靠底层 C 扩展。
- Python 对于硬件编程和实时特性较弱。实时控制逻辑尽可能封装在 C 语言驱动程序中。
- Python 的硬件扩展库还未标准化。
- 合理使用 Python，可以降低通信堆栈开发难度，加快应用开发速度。

第 6 章

嵌入式 Python 虚拟机

本章重点介绍各种运行在嵌入式系统的 Python 解释器（虚拟机）。实际上， Python 在嵌入式系统中早已经存在了多年。前面提到过，由于它的开源本质，Python 已经被移植在许多平台上。在这些平台中，Palm OS、QNX、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE、PocketPC、Symbian 和 Android 都可以归类到嵌入式系统中。

6.1 嵌入式高级语言平台大荟萃

目前嵌入式系统中的主导开发语言是 C/C++和 Java。但是开发语言远不止这些，随着硬件资源成本的快速下降，各类动态语言（例如：Python、Lua、JavaScript、C#）开始占据嵌入式系统，包括 MCU 在内的开发。在高性能嵌入式 CPU 平台中，几乎和桌面一样，采用了 Ruby、shell、Jython、Golang 等语言。本章着重介绍 Python 在各个平台以及嵌入式 MCU 中的普及情况。

6.1.1 高级语言与二次开发

使用虚拟机（无论是脚本还是字节码）的最大价值是，**提供了用户的二次开发能力**。

我们最初能够看到的高级语言运行环境是 Intel 的 BASIC-51。当时 Intel 为了推广 8051 架构，推出了 8051 专用的 BASIC 运行环境。这个版本的 BASIC 解释器，采用汇编语言编写。

不过，无法固化在 ROM 中的 BASIC 仅仅是好玩而已。笔者并没有深入体会底层汇编语言和高层应用语言分离的意义。随着系统的复杂性日渐增加，传统语言在日常开发中的矛盾越来越突出，如：

- 复杂系统导致超长编译时间；
- 开发包含闭源软件库导致调试困难；
- 用户应用程序频繁修改导致系统整体开发时间拉长，并导致工程延误。

在评估过一些高级语言的嵌入式开发后，笔者深刻体会到了在嵌入式中引入二次开发的必要性。

6.1.1.1 超长编译时间

如果是嵌入式 MCU 系统，那么应用程序、RTOS 和通信堆栈都是编译链接在一起的。即便是嵌入式 Linux，整体的开发流程也是不断地修改 C/C++代码、反复交叉编译。一般 Linux 内核编译次数少于应用层的编译次数。但是许多情况下，某些组件需要剪裁内核并重新编译。

如果系统复杂度非常高，编译链接时间就会很长。笔者曾经支持过的飞利浦 DTV520 平台，底层采用 Linux，内核编译时间算是短的。但是其高层 UI 和业务控制逻辑编译一次需要四个小时。基本都是修改一次，等上半天。所以工程师修改源码变得非常谨慎。这种漫长的编译过程，显然在交付和迭代速度上存在问题。

解决方法之一：提升编译主机性能。笔者当时应对这种情况的方法是使用高配计算机用于编译，所有源码本地编辑，FTP 上传，采用远程登录方式进行编译。采用这种方式可以缩短一部分编译时间。不过，高配计算机提供的性能提高很有限。

解决方法之二：分布式编译池。笔者评估了 distcc 分布式编译，使用多台计算机进行编译。这的确可以降低编译时间，而且支持交叉编译。但是 distcc 依赖于 Make 工具，如果采用了 Make 之外的构建工具，则需要先剥离出来，然后再进行 C/C++编译。

解决方法之三：使用脚本化语言虚拟机环境。Python 虽然也需要编译成 Byte Code，但是却又具备脚本化语言的特点。C/C++使用 makefile 进行管理，修改一个 C/C++文件的编译时间不长，但修改一个头文件有可能会涉及一批文件需要重新编译；而 Python 等脚本化语言却可以直接运行源码。

6.1.1.2 闭源库的调试

在商业化开发中，经常性地包含第三方闭源软件库。比如桌面应用中常见的 DLL 和头文件。在嵌入式开发中，也有这种现象，许多软件供应商仅提供 Lib 和头文件。因为这种黑箱方法往往会造成调试困难，所以大多数开发者都不喜欢这种开发模式。

MCU 程序是静态编译和源码调试的。一旦代码运行到库文件范围内，开发者是看不到底层源代码的。软件库中的算法实现、何时返回调用函数，此类信息如果缺乏及时的沟通和技术支持，开发者就会有挫败感，整个开发周期也因此会被人为地延长。现在，许多 BLE 模块也是如此，将 BLE 堆栈以 ROM API 的形式交付，但是看不到源码。

其实，如果平台内置虚拟机运行环境，则完全没有必要这么做。这道理看手机 APP 开发就很好理解了：如果 Android 开发需要把虚拟机、UI 框架和用户应用一起编译，那会是什么结果。就算排除调试困难的因素，应用层开发者也不会对底层细节感兴趣。

所以，开发者需要的是操作系统对于硬件的抽象，而不是简单、粗暴地封闭源码。

6.1.1.3　应用层频繁修改

1999 年 emWare 公司展示的一种嵌入式汇编语言虚拟机，让笔者大开眼界。

当时 emWare 设计是在多种架构 MCU 上构建一种虚拟机运行环境，虚拟机执行自定义的汇编码。用 Perl 脚本做简单汇编后装入 EEPROM，并由虚拟机逐字节执行。这种自定义非常灵活，可以针对应用进行定制。举例来说，一般汇编码是“正交”的，也就是不能够有重复，不可以引发解码逻辑的误解和冲突，而且宽度往往是一定的。但是，这种完全定制的汇编语言却可以做成没有特定长度限制的汇编码。假设我们需要打印字符串“Hello”，串口打印指令对应的二进制指令 0x55，那么完整的汇编指令就是 0x55,0x68,0x65,0x6C,0x6F,0x00。此外，emWare 虚拟机定义了一组虚拟寄存器，作为操作数的中间存储单元。

最初引入这种虚拟机的目的是利用这种虚拟机提供一种嵌入式 Web 服务器引擎。用户可以用这种虚拟机代码设计自定义网页内容，下载并运行。这种设计分离了应用层和底层架构，为应用开发提供了更快的迭代方式。

无独有偶，Lua 语言就是基于自定义寄存器的虚拟机，而且 Lua 具备完整的高级语言特性。所以，在经过多年开发后，Lua 也成为一种主流的虚拟机高级语言。

利用虚拟机将应用层分离出来，让应用层开发者专注于自己的应用，而非底层细节，这让人印象深刻。这也是现在各种高级语言开始在嵌入式开发中崭露头角的原因之一。

6.1.1.4　智能硬件 2.0

虚拟机除了能够解决开发中的矛盾，更加重要的是迎合了可编程设备的需求。

还记得 Web 2.0 这个术语吗？这是一种利用 Web 平台，用户创建内容为特点的互联网产品时代，以区别于网站编辑内容的 Web 1.0 模式。

从技术角度来看，社交化、分享、开发平台引出了最重要的技术革新：XML、JSON 和 Web 服务。这意味着任意对象都可以被抽象，并通过 API 提供 Web 服务。同时，通过 Web 服务聚合（Mash-up）可以产生新的服务。最著名的谷歌地图 API 和其他 LBS 服务整合就是典型例子。Programmable Web 到目前依然是快速迭代和演变的领域。

将网络服务的概念推广到物联网和智能硬件领域，或许可以衍生出 Device as a Service 的概念。智能硬件和物联网将提供某种程度上的 Mash-up 能力。这意味着，任何设备不仅仅具备联网能力，还应该具备某种可编程能力。这种可编程能力意味着：

- 标准化通信接口（如 IPv6）；
- 标准化协议（REST/JSON/XML/CoAP/MQTT）；
- 鉴权（TLS/D-TLS）；
- 动态更新（配置或者应用程序下载，XML/JSON/Java/Python）。

让我们继续列举几个简单的例子。

1．智能照明

Philips Hue 是一款比较出名的智能硬件产品。除了基础照明功能，通过手机定位功能，Hue 还可根据用户状态自动开关灯，改变灯光颜色。通过设定灯光定时提醒，Hue 可以让人们每天的生活更加有规律。

通过连接互联网，Hue 还可以实现更多智能应用，包括显示天气状况、比赛结果、股票信息、电子邮件等；人们可以根据自己的需要进行任意设置。例如，可以设置灯光颜色变化来表示将要下雨，设置灯光慢慢变亮表示太阳正在升起，甚至可以在自己喜欢的球队赢得比分时让灯光闪烁球队的标志颜色。有的能够用灯光的变化来显示心跳；有的能够让人们通过声音来控制灯光；还有的能和电视屏幕的变化同步。这些都可以通过 IFTTT 网络功能来轻松实现。用户也可以通过该平台分享其个性化的灯光配置。现在已经有超过 40 个新的应用程序被开发出来。

Hue 单组、多组和群组配置，组合起来后可以出现更多的变化。只是因为 Hue 的价格偏贵，反而压缩了其普及的程度。实际上在飞利浦公司内部，多年前就安装了可以根据不同的办公室场景进行切换的照明系统。办公室照明和其他设备可以根据会议、讲演、聚焦和日常模式进行切换。不过作为一种专业照明产品，其并不为太多人所知晓。这是因为这些产品就像 Web 1.0 一样，是由专家预设的模式，不具备可编程性。实际上，Hue 提供的核心功能 API 只有开关、RGB 配色而已，但其却通过网络 Mash-up 应用聚合后，开发出多种用途，成为消费照明领域的里程碑产品。

类似的单品还有 Google Nest 等。小一点儿的产品还有 LED 车尾灯、智能手环、智能手表。它们都可以通过设备与应用的 Mash-up 形成新的生态圈。

2．安防系统整合

安防系统由于和安全相关，所以不会成为一种消费化的产品线。安防系统虽然可以依赖于标准 IP 通信，提供标准化的应用层协议和控制，但是从系统集成角度来看，却需要大量的定制和后续服务。

笔者最近在和一位网友讨论“互联网思维”小区门禁系统中挖掘了此类需求。不同的小区，各自地理分布特性、RFID 兼容性、BLE 普及率、Wi-Fi 普及率、微信硬件等，以及与业主俱乐部、停车场的整合程度不同，导致系统业务逻辑需求多变，而且无法固定下来。最后笔者推荐实施了一款以 MPU 为主的平台，兼备 RFID/NFC、BLE、Wi-Fi、LTE 及二维码模块等多种连接能力的兼容型读卡器。读卡器与系统服务器之间采用 REST API 进行整合。此外，还可以利用移动互联网、微信、APP 完成互联网营销。这种设计不仅可以满足不同安防需求；即使发生房产租售和物业公司更换，也可以保留原有的基础建设，保护现有的投资。

和安防系统结构类似的系统还有楼宇自动化、消防自动化、工业自动等，也可以采用类似的方法实现。

3．频繁更新应用的设备

笔者认为在工业 HMI、智能手表和智能手环等具备基础 UI 特性的产品中，可能存在修改应用层固件的需求。因为在此类产品中，将更多控制权交给最靠近用户的开发者才可以满足定制和多样化需求。

根据这点认知，我们可以实施以下设计：

- 可编程 LED 条屏。假设社区所有的条屏均可以编程的话，是否可以诞生新型的应用和商业模式：LED Bar as a Service？
- 可编程的楼宇照明系统、可编程的景观灯、可编程的喷水池。

在以上若干例子的启发下，读者是否可以自己开开“脑洞”？

接下来介绍一下嵌入式行业中可以用得上的高级语言平台。评价一个嵌入式行业可用的高级语言，不能够被语言本身的优势所迷惑，还要综合考虑配套工具和软件包以及社区支持方面的完整度。只有这些都具备，才能够形成一个完整的生态链条。

6.1.2　BASIC

前面已经提到了 Intel 的 BASIC-51。它是一种解释型的 BASIC，即针对用户的 BASIC 代码，逐句读进来，解析后执行。其工作原理类似于 ROM BASIC 8086。

如果读者有兴趣，可以下载 Intel 源码研究一下，移植到其他应用平台上。不过，最好使用 C 语言来构建 BASIC 解释器。在网络上也有一些开源 BASIC 解释器的源码和教程，大家可以拿来玩玩。Adam Dunkel 提供了一个 uBASIC（http://dunkels.com/adam/ubasic/）。Adam 同时也是 LwIP/uIP/Contiki 等一系列开源组件的开发者。

6.1.3　Java

Java 是除了 C/C++之外，市场占有率最高的嵌入式高级语言。Java 设计之初就是针对嵌入式开发的。JavaCard、移动电话 J2ME、蓝光 DVD J2ME、Android 手机、平板电脑、机顶盒和电视，都是 Java 语言的应用场景。J2EE 至今为止还是许多企业应用和 Web 站点的首选平台。

在笔者的使用经验中，大多数嵌入式 Java 虚拟机都是商业版。虽然可以找到免费的 Java 虚拟机，但是其往往缺乏源码，而且往往也禁止商业用途。第三方 Java 虚拟机无论开源与否，大都自称 Clean room implementation。这种声明特指独立实现的逆向工程，而非简单的复制，其主要用于回避版权纠纷。在编写本书之前，笔者特地重新了解了 Java 语言的近况。

6.1.3.1　Java 的各种实现

Sun 已经在 2006 年 11 月 13 日，在 GPLV2 协议下将 Java 正式作为开源软件（包括 J2SE 和 J2ME）。所以，Java 开源已经 10 年了。在 open-open.com 网站罗列了一个很长的开源和闭源

Java 虚拟机列表。以下是各种嵌入式版本 Java。

免费和开源

- Avian，一个小巧的可嵌入 Java VM 和 classpath，使用 JIT 编译。
- HaikuVM，使用 leJOS 运行环境，针对 Atmel AVR 及 Arduino 和其他单片机。
- Juice，JavaME 实验性 JVM。
- JwiK，面对 8 位无线应用设计开源 Java VM。
- leJOS，机器人套装，面对乐高 Mindstroms 系列可编程机器人提供的替换固件，可以为 RCX/NXT 机器人提供 Java 编程环境。
- Mika VM，为嵌入式设备提供的跨平台虚拟机，采用 BSD 许可证。
- NanoVM，为 Asuro 机器人内置 Atmel AVT ATmega8 而开发的虚拟机，可以移植到其他 AVR 系统中去。
- Squawk VM，Java ME VM，针对嵌入式系统和小型设备，跨平台，并采用 GPL 协议。
- SuperWaba，类似 Java 的移动设备虚拟机，采用 GPL 协议。其被 TotalCross 所替代。
- TakaTuka，针对无线传感器网络设备而设计的 Java 虚拟机，采用 GPL 协议。
- VM02，为苹果 II 型计算机提供 Java 兼容环境。
- Wonka VM，为 Acunia 的 ARM 硬件开发的嵌入式虚拟机，采用 BSD 许可证。其被 Mika VM 所替代。

收费

- Apogee，基于 IBM J9 和 Apache Harmony 类库提供嵌入式 Java，主要针对运行 Linux、LynxOS、Windows CE 操作系统的 x86、ARM、MIPS、PowerPC 设备。
- JBed（Esmertec），附带多媒体能力的嵌入式 Java。
- Jamaica VM（aicas），一种嵌入式系统的硬实时 Java VM。
- JBlend（Aplix），是 Java ME 的一个实现。
- MicroJVM（IS2T，Industrial Smart Software Technology），专门针对包括硬实时资源受限在内的嵌入式系统，如 ARM/AVR/PPC/MIPS 提供各类虚拟机。
- PERC（Aonix/Atego），嵌入式系统的实时 Java 虚拟机。
- PreonVM（Virtenio），面向嵌入式和小型设备的虚拟机。

从技术上来说，Java 是非常成熟的，Oracle 页面中有许多可以用于嵌入式系统的产品：J2SE/J2ME/Java Embedded/Java Card/Java TV。但许多 Java 应用有着各自的问题：

- J2ME 主导的功能手机时代一去不复返；
- 蓝光 DVD 一直不温不火，带 J2ME 的 BD 碟片太少，以至于笔者很难找到合适的样本；
- JavaCard 更是被各类 NDA 包裹得严严实实；
- 在台式 Linux 上，Java 很流行，但是 Python 开发起来更加容易；

- 单独运行 Oracle/Sun 的官方 Java 的主流设备较难寻找。

不过我们还需要看到：

- Java 已经部分归属于开源软件；
- Android 实际上是基于 Dalvik VM 的 Java 变种；
- 基于 MCU 的 Java 虚拟机还可以在物联网应用中占领一定的市场份额。

在 Oracle 的下载页中，Java Embedded 针对树莓派和 Freescale K64F 提供了二进制虚拟机，可供下载使用。

6.1.3.2　Motorola MODEM 与嵌入式 Java

在实际工程中，笔者使用过一款 Motorola（摩托罗拉）的 GPRS/EDGE 模块 G24，用于工业水表的远程抄表系统。其技术特性之一就是内置 J2ME 支持。G24 是 2006 年出品的 GPRS/EDGE MODEM 模块，如图 6-1 所示：

- 支持 J2ME CLDC（G24-J），使用 Sun 的标准工具链开发；
- ARM7TDMI CPU，用户可用存储器为 10MB Flash ROM、1.8MB RAM；
- 内置 TCP/IP 堆栈；
- 支持电子邮件；
- 支持 HTTP 协议；
- 支持 FOTA 远程固件更新；
- 支持 IBM MQTT 协议；
- 支持 SSL 协议；
- 两路 UART、一路 USB 和一路 I2C；
- 无须外接 MCU 做主控制器，支持 self-host 模式。

图 6-1　Motorola G24 GPRS/EDGE MODEM

G24 与同时代的 MODEM 相比，技术非常领先。即使与现在的 MODEM 相比，其依然是领先的。2006 年的产品已经内置 MQTT 协议和 SSL 协议，而现在国内主流的 MODEM 都只有 TCP/HTTP 协议而已。但随着 Motorola 的没落和股权的换手，相关技术资料已经无法找到。眼下只有通过搜索引擎来收集此类资料了。

有兴趣的读者可以在本章延伸阅读部分中找到 G24 的 Java 宣传册，并在硬件手册中查看到技术细节。目前在网络上可以采购到库存的 G24 模块，但是其价格非常贵。

6.1.4 Lua

Lua 是轻量级的嵌入式脚本语言，其于 1993 年首次面世。所谓嵌入式脚本是指可以嵌入其他语言中的语言，当然在嵌入式系统中也可以使用。

Lua 虚拟机是基于寄存器结构的，所以其虚拟机实现可以很简单。eLua 就是嵌入式系统的 Lua 平台。可以支持大多数的微处理器架构，如 LPC1768/STM32F429/MK20DX256/CC3200 等。

eLua 号称最轻省，可以编译出小于 200KB 的平台，其系统架构如图 6-2 所示。但是大部分 MCU 都是按照 16/32/64/128/256 翻倍的，所以 ROM 至少也是 256KB 版本的。

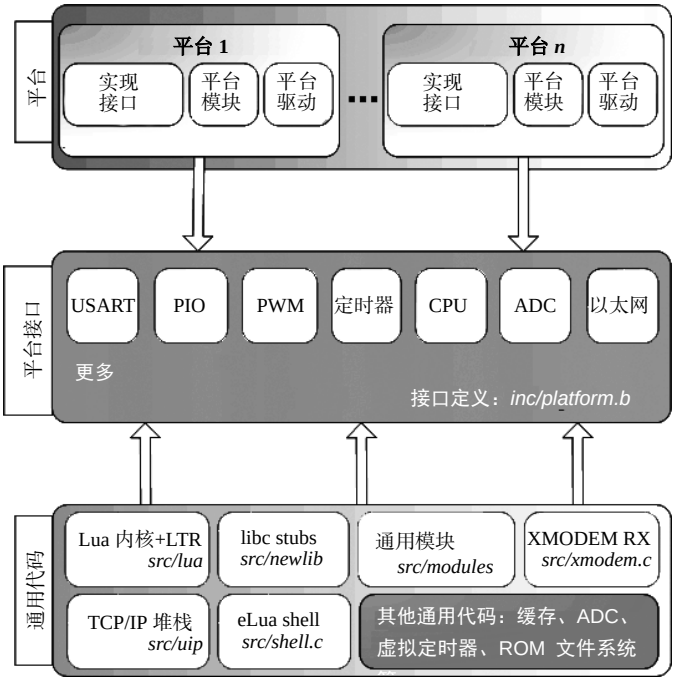


图 6-2 eLua 系统架构图

在编写本书的过程中，不断有读者问为何不写 eLua 应用书籍的问题，让笔者觉得这至少反映了一些开发者的倾向、态度以及应用趋势。

6.1.5 JavaScript

JavaScript 从一个玩具语言变成了生产环境语言，并有全栈开发的趋势，这不仅仅指 Web 开发的前端和后端，也包括嵌入式系统。JavaScript 支持的嵌入式平台如下：

- Esprino，运行 JavaScript 的子集；

- Tessel，支持 JavaScript 的子集，首先将 JavaScript 翻译成 Lua，之后在 MCU 上执行；
- Kinoma，来自 Marvell 的 IoT 套件，支持完整版 JavaScript；
- Curie，来自 Intel/Arduino 101，支持 JavaScript 编程。

所以，JavaScript 如果加大推广力度，在嵌入式领域中可以有很大作为。实际上，已经有许多第三方设备板载虚拟机都实现了 JavaScript 的子集。

6.1.6　.NET

netdruino 和 FEZ，这两个开放平台使用 C#。现在，在嵌入式设备中，C# / .NET 真的不算主流。其原因和技术无关，更多的是与使用成本、许可证、商业模式有关。微软开始拥抱开源平台, C#或许会成为开源工程的选择之一。

6.2　前一代 Python 虚拟机

自从 iOS 和 Android 占据了智能机市场后，早前的“智能机”就被人们淡忘了。但其实在前一代的主流智能手机平台中，Python 也曾经是玩机一族的最爱。由于其开源的特点，所以 Python 在许多平台上都有过应用。我们先看看移动通信模块中的 Python，这个属于深嵌入式的 Python。

6.2.1　Telit GPRS 模块

Telit 是意大利的蜂窝数据通信模块供应商。到目前为止，这也是唯一一家在商业产品线中使用嵌入式 Python 的商业机构。其官网上相当多的产品线都内置 Python 脚本支持，如图 6-3 所示的 GE/GC864 系列是国内较为常见的型号，也支持 Python 开发。

图 6-3　Telit GE864-GPS 嵌入式蜂窝数据模块

针对 GSM/GPRS/GPS，Telit Python 提供了多个基础 Python 包，如表 6-1 所列。

表 6-1　Telit Python 包

定制模块包	说　　明
MDM	Python 与通信模块间 AT 指令接口
MDM2	Python 与通信模块间 AT 指令第二接口
SER	Python 与通信串口 ASC0 间接口
SER2	Python 与通信内部串口 ASC1 间接口
GPIO	Python 的 GPIO 包
MOD	Python 的其他杂项包，包括时钟、看门狗、功耗管理、Base64 codec
IIC	Python 的 I2C 总线包，该总线可以映射到任意 GPIO
SPI	Python 的 SPI 总线包，该总线可以映射到任意 GPIO
GPS	Python 与模块内部 GPS 控制器的接口

Telit 支持 Python 脚本引擎版本 V1.5.2+，从图 6-4 的示意图中可以看出，Telit 的 Python 运行环境与底层 MODEM 驱动程序共享 RAM/ROM/CPU 资源，所以存在一些限制。

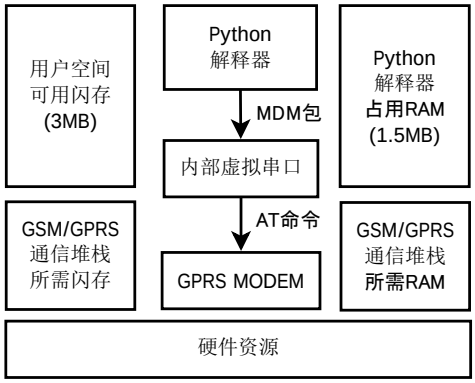


图 6-4　Telit Python 运行环境示意图

Telit Python 的限制如下：

- 1MB 用户脚本存储空间，最多为 2MB。
- Python 引擎需要占用 1.2MB RAM。
- 每个对象最多为 16KB RAM。
- Python 脚本每 50ms 中断一次，且不可以干扰 GSM/GPRS 标准操作。
- GPIO 轮询频率小于 100Hz。
- I2C 和 SPI 总线速率在 10kbps 到 20kbps 之间。

Telit 是一个完整的数字控制系统，仅缺乏 ADC 和 PWM 驱动，而且 GPIO、I2C、SPI 还有

性能提升空间。可惜 Telit 作为一家商业机构，并没有将其源码分享出来。本人推荐使用开源 Python VM 用于驱动各类物联网 MODEM。

6.2.2 Symbian

Nokia（诺基亚）的 Symbian S60 是 Nokia 主推的智能机平台。除了运行 C/C++、J2ME 程序，还可以运行 Python。虽然 Python 是后起之秀，但其易用性却赢得了许多开发者的青睐。开发者可以通过蓝牙连接手机和 PC 进行开发。Python S60 平台也将手机的大多数功能开放给 Python 开发者使用。

Mobile Python, Rapid Prototyping of Applications on the Mobile Platform（Wiley 出版社，作者为 Juergen Scheible、Ville Tuulos）介绍了 Python for S60 的可用资源：包括 GUI、SMS、多媒体、文件系统、LBS、移动网络和网络服务。读者可以找机会在网上淘个经典 Nokia 手机来试试看。

6.2.3 Windows CE

在 sourceforge 上的一个开源项目：Python for Windows CE。虽然此类设备已经被淘汰，但可以作为嵌入式 Windows 平台上的 Python 虚拟机基础。

微软官方已经不再支持 Windows CE 平台，Windows 平台的物联网发展要看 Windows 10 IoT 的普及情况。Windows 10 IoT 已经适配 ARM 平台的树莓派和 x86 平台的 Galieo,，同样支持 Python 和其他编程语言。

6.2.4 OpenMoko

OpenMoko 工程的目的是建立第一个开源的移动通信操作平台。此平台的 GUI 运行在 X Server 之上，可以运行大多数 X 应用程序。该项目由中国台湾 FIC 支持，并提供运行 OpenMoko 的硬件、Neo1973 和 FreeRunner。OpenMoko 上运行的软件大多数来自 OpenEmbedded，包括 Python 运行环境。

OpenEmbedded 的主要目的如下：

- 针对多个平台提供经过交叉编译过的软件包；
- 处理不同的硬件架构，并针对这些架构发布软件包。

目前，OpenEmbedded 的许多组件依然可以在大量的廉价 Linux 主控板中使用。随着 Linux 与各类 SBC 的发展，越来越多针对 ARM 的完整版 Linux 已经出现，已经不需要交叉编译。不过，OpenEmbedded 依然是低成本 Linux 设备，如路由器、Web 摄像头的重要软件来源之一。

6.3 深嵌入式 Python 平台

除了上面提到的与移动计算平台密切关联的 Python 虚拟机，我们即将介绍深嵌入式系统中的 Python 虚拟机。大多数嵌入式系统可能没有显示系统，最多采用串口与外界通信。此类系统资源和计算能力更加受到限制。但是基于嵌入式和物联网的本质，Python 除了支持基本的硬件外设之外，还可以在物联网通信和高层应用中发挥高级语言优势。

6.3.1 LEGO EV3

LEGO EV3，如图 6-5 所示，是乐高公司头脑风暴（Mind Storms）系列机器人中的第三代产品。采用 Atmel 300MHz ARM926 处理器、16MB Flash ROM 和 64MB RAM。除了官方固件之外，EV3 还有 dev-ev3 固件。dev-ev3 是基于 Debian ARM 的操作系统。所以，开发者可以使用 Linux 中的各个编程语言，包括 Python 语言来控制机器人。

图 6-5 乐高公司 EV3 头脑风暴系列机器人

LEGO EV3 证明了一件事情：即使是最小的 Linux 发行版，只要有外部 SD 卡插槽，都可以将最小的 Linux 升级为完整版 Linux，以实现更多的功能。而基于 Linux +动态语言的编程模式将彻底改变原有的交叉编译开发模式。许多芯片开发商之所以不愿意将自己的老旧 ARM9/MIPS 平台升级到最新的操作系统，主要还是因为缺乏商务上的动力。

6.3.2 TinyPy

TinyPy 是一个很小的嵌入式 Python，可以在小于 64KB 的 ROM 中运行。TinyPy 的作者 Phil Hassey 将其作为一个提高自身 Python/C 编程能力的途径。TinyPy 实现了虚拟机的基本特性，我们可以研究一下其编译、运行环境的处理和实现。很遗憾的是，该项目目前不再更新了。所以

TinyPy 适合作为一个玩具/教学性质的虚拟机。

实际上，各类嵌入式 Python 不仅能够支持大部分 Python 特性，而且还向 Python 社区提供了资源受限情况下的大量优化设计，这也为日后桌面 Python 系统的性能提升提供了很好的实例，并为社区做出了自己的贡献。

6.3.3　嵌入式 Python 的局限

前面提到桌面版 Python 的局限是速度和代码开放度。在嵌入式系统中，则还有更多的限制，主要体现在 RAM 和 ROM 资源相对较少。

面向对象编程语言往往比面向控制编程语言要耗费资源。Python 比 C++/Java 更加抽象，在 Python 中，包括内置类型，一切都是对象（PyObject）。其实无论是 C、C++、Python 还是其他语言移植到嵌入式系统中，都会经历系统适配优化的过程。但是与 C/C++不同，所有基本数据类型在 Python 中都是对象。这意味着一个无符号整数在 Python 可能远不止一个字节。这对嵌入式 Python 的开发者是巨大的挑战。

根据 MicroPython 作者最初的评估，操作一个简单字符串就需要 4KB RAM。如果要操作其他片上资源，或者使用列表类型，那么耗费的存储器和计算资源就更多了。所以，嵌入式平台上运行的 Python 必须做剪裁或者某种程度的优化定制。这也要求应用开发者在编写代码时需要时刻记得这一点。

接下来，笔者将在 STM32 平台上测试并介绍三种深嵌入式 Python 平台：

- PyMite，基于 STM32F103RB；
- MicroPython，基于 STM32F401RE；
- VIPER/Zerynth，易于 STM32F401RE。

这样会让读者能够充分了解这些面向微控制器的嵌入式 Python 平台，希望能够打破 Python 在嵌入式领域无法发挥语言能力的思维定式。

在介绍完这些深嵌入式 Python 之后，会针对 Linux 最小系统中的 Python 进行讨论。完整版 Linux，无论是 x86 还是 ARM/MIPS 平台均已经内置 Python 支持，所以并不需要特别讨论其虚拟机的实现。但 Linux 最小系统往往只有 busybox，传统的方式是将 CPython 进行交叉编译剪裁定制。这对于开发者的要求较高。嵌入式 Linux 不仅可以使用交叉编译的官方 CPython，也可使用交叉编译的 MicroPython 的 UNIX 版本和 PyMite 的 desktop 版本。由于后两者已经为交叉编译做了处理，因此可以简化交叉编译的过程。

6.4 PyMite

PyMite 是一种羽量级 Python 翻译器。PyMite 的开发者 Dean Hall 曾经在 Motorola 工作过，开发维护过 J2ME 虚拟机。出于个人爱好设计并推出了 PyMite，现在改名为 Python-on-a-Chip（p14p），以 GPL 许可证方式发布。

PyMite 支持的特性如下：

- 多重继承的类；
- 生成器，支持迭代器、表达式和协程；
- 支持“+”号合并字符串；
- Python 2.6 编译器和字节码；
- 整数和浮点数的反引号操作符（s='x'）；
- 字符格式化符号%，支持%d %s %f；
- 闭包，允许带参数装饰器；
- ByteArray 类，packet = bytearray(128); b = bytearray（b"abc"）；
- 改进 IPM（Interactive PyMite），可以理解为 PyMite 的 REPL；
- 改进型 GC 算法；
- 改进异常回溯的可读性；
- 将原始函数表设置为“const”参数类型，以节省 RAM；
- 将 iter 作为内置函数；
- 用户脚本可以存储在 ROM、RAM、EEPROM 甚至外部串行 EEPROM 中。

6.4.1 硬件平台

PyMite 是一种针对中低端 8~32 位嵌入式控制器的超迷你 Python。其目前可以支持的硬件平台列在表 6-2 中。

表 6-2 PyMite 硬件一览表

开发板	微控制器	内　　核	RAM（KB）	ROM（B）	频率（MHz）
Arduino MEGA	Atmel ATmega1280	AVR	8	128	16
AT91SAM7S-EK	AT91SAM7S64	ARM7TDMI	16	64	55
AVR	Atmel AVR	AVR	4	64	8~16
x86	PC	x86	无限制	无限制	
econotag	FSL MC13224	ARM7TDMI	96	80	26
mbed	NXP LPC1768	Cortex-M3	64	512	100

续表

开发板	微控制器	内　　核	RAM（KB）	ROM（B）	频率（MHz）
MMB103	Atmel ATmega103	AVR	4	128	16
PIC24	Microchip	PIC24/dsPIC		16	256
ET-STM32	STM32F103RE	Cortex-M3	64	512	72
Teensy	Atmel ATmega1280	AVR	8	128	16

PyMite 重新实现了 Python 翻译器。其目标运行平台从 8 位机开始，最小仅需 64KB ROM、4KB RAM，不过文档中推荐最小可用 RAM 为 8KB。PyMite 支持 Python 2.6 语法子集和字节码的子集。PyMite 可以在桌面系统中进行交叉编译、测试和运行。笔者认为，这个特性使得 PyMite 可以运行于嵌入式 Linux 中。

由于是重新编写的 CPython，因此 PyMite 的许多特性适用于资源受限系统。例如：启动时无须堆栈，支持原生 C 函数。在这一点上，其倒是和 Lua 有点儿像。

该项目最初的硬件平台是 AVR，这充分显示了其虚拟机微型化的能力。不过真正可实用的工程或者产品，还应该在 ARM 平台上，RAM 配置为 16~256KB，而 ROM 配置为 128~512KB。由于 Python 的运行环境主要在 RAM 中，所以需要使用内置 RAM 较大的微控制器品种。

6.4.2　维护者

PyMite 代码最初托管在 Google Code。一方面，在国内无法正常访问 Google Code；另外一方面，Google Code 面临 GitHub 等社交编程网站的冲击，已经宣布关闭。目前其所有代码均为只读状态，以供用户下载。

虽然 mbed 网站上有专门的维基页面，但是 Dean Hall 并没有给出最终的 PyMite on mbed 代码，只是告诉我们如何移植代码。这说明 Dean Hall 已经不再维护 PyMite。但是其源码依然是最简单的 Python 虚拟机之一。事无巨细，方方面面都已经开放了。配合主流 MCU，价格便宜，可以用来试试手。

此外，日本开发者 Norimasa Okamoto 维护着 PyMite 的 mbed 版本 pymbed。

6.4.3　pymbed 分支

前面提到的 ARM mbed 平台，通过 C/C++可以实现更高层面的抽象。现有的 mbed 平台非常活跃，可以使用大量的第三方库和中间件。如果将这些第三方库进行 Python 封装后，可以面向产品提供现成的设计。

Okamoto 一口气将 PyMite 移植到了 mbed 支持的十多款 MCU 中，具体型号列在表 6-3 中。为了与原始 PyMite 有所区别，Okamoto 提供的 PyMite 版本被称为 pymbed，即 PyMite for mbed。

表 6-3　PyMite 支持的 ARM mbed MCU 列表

MCU	内　　核	RAM	Flash ROM	外　　设
LPC1768	M3，96MHz	32KB	512KB	NIC/USB Host/Device
LPC1549	M3，72MHz	36KB	256KB	Standard MCU
LPC11U68	M0+，50MHz	36KB	256KB	USB Device
LPC4088	M4，120MHz	96KB	512KB	NIC/USB Host/Device x 2
KL25Z	M0+，48MHz	16KB	128KB	USB-OTG
KL46Z	M0+，48MHz	32KB	256KB	USB-OTG
STM32F103RB	M3，72MHz	20KB	128KB	USB Device/USB
STM32L152RE	M3，32MHz	80KB	512KB	LCD/OpAmp/USB
STM32F401RE	M4，84MHz	96KB	512KB	USB-OTG,SDIO
STM32F411RE	M4，100MHz	128KB	512KB	USB-OTG,SDIO

在上述处理器中，推荐以 LPC1768/KL25Z/STM32F103RB 作为起点。这些都是常用 MCU 型号，容易采购。总的来说，笔者觉得 PyMite/pymbed 可以是嵌入式 Python 软件开发的起点。

Okamoto 在 mbed 上留下的作品说明他是一位特别专注于开源硬件，并能够充分利用各种资源的程序员。除了 pymbed，他还为 mbed 的许多硬件提供了多种工具：IAP、USB 报文分析器、摄像头主机驱动，甚至包括某些 MCU 如 LPC812、LPC1114 的仿真软件。

同时，为了避免开发者过多地纠缠于 GCC 编译问题，他在 Google appengine 上提供了免费的在线开发环境，包括编辑器、编译器，如图 6-6 所示。这种在线编译模式和 mbed 在线编译是一样的思路。

```
import mbed
import sys

myled = mbed.DigitalOut('LED1')
cnt = 0

sys.wait(2000)
print "Hello world, from pymed.\r\n"
while True:
    myled.write(1)
    sys.wait(200)
    print "cnt: %d\r\n"%(cnt)
    cnt += 1
    myled.write(0)
    sys.wait(200)
```

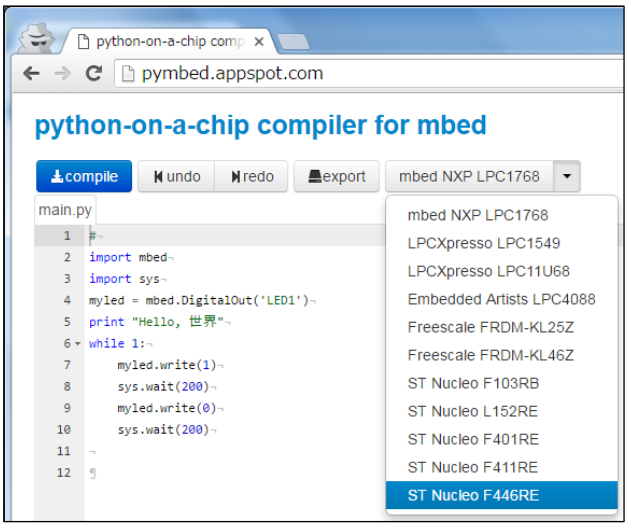


图 6-6　pymbed 在线开发环境

笔者选择 STM32F103RB，测试了一下上面所附的简单代码。单击“compile”按钮后，将编译后的 bin 文件下载到 NUCLEO-F103RB 中去。用 TeraTerm 连接后，串口打印出预先设定的内容。如果字符串中有 UTF-8 编码的汉字，在终端中也可以正常显示。如果看到乱码，请检查波特率是否为 115200bps，以及 TeraTerm 的文字编码和字体设置是否有问题。

6.4.4　开发现状

现有的 PyMite/pymbed，无论是维护者还是使用者都不活跃，原因如下。

- 编译过程不方便：PyMite 的用户脚本需要先在主机编译成字节码，然后转换成 C 语言字节数组，还需要做一次 C 语言编译。换而言之是两次编译过程，这一点在 MCU 开发中不是很灵活。
- 封装层次低：PyMite 所支持的硬件 I/O 层级较低，都是 DIO/ADC/PWM/Mem/I2C/ SPI/ UART 等标准 I/O，与 C/C++代码相比缺乏速度优势。Okamoto 还提供了一些 Python 的 LCD 例子，但是体现不出 Python 的语言优势。
- 扩展不够灵活：PyMite 是通过原生 C 函数来实现系统扩展的。由于嵌入式系统的多样性，许多复杂的设备都需要定制后将这些 C/C++ API 暴露给 Python 虚拟机。这方面的工作要系统开发者自己承担。
- 缺乏中间件和扩展库：由于扩展和高级组件的缺乏，物联网常见的库也没有得到发展，比如 FAT 文件系统、TCP/IP、Web、CoAP、6LowPAN 等。
- 缺乏足够的演示代码和技术支持：维护者不积极，开发者也无法聚拢人气。

现在 PyMite 的系统抽象级别比较尴尬，既没有足够底层基于寄存器的 API，又没有足够抽象的中间件扩展。中间件定制又需要针对特定硬件进行定制，需要具备 C/C++驱动底层硬件的开发经验。熟悉 C/C++的开发者往往不能理解 Python 的语言优势；熟悉 Web/App 的开发者往往又无力针对硬件进行定制。所以这个系统变成两端都无法继续推进的平台了。作者没有持续迭代，构建开发者生态，与开发者良性互动，形成合力，这造成 PyMite 鲜为人知。

许多开发者包括笔者本人兴冲冲而来，遇到困难无法自己解决，必须依靠维护者或者社区论坛来寻求支持，如果这一点没有得到满足，往往就会停滞不前。

笔者认为 PyMite/pymbed 需要在以下几点改进：

- 提供寄存器级别的 API，许多硬件的扩展可以在 Python 级别进行。
- 在线编译需要从 Google appengine 平台移植到通用的 Web 框架，比如 Flask。
- 支持字节码下载执行，Python 编译的字节码直接烧录在 Flash ROM 或者外部 EEPROM 中。
- 充分的代码和应用支持。就算没有技术支持，也应该提供一些完整的项目。
- 整合网络资源。现阶段 PyMite、pymbed 的资源分散，需要整合在一个网站中。
- 缺乏原生 REPL。PyMite 的 IPM 就是 REPL，但是需要用户在 main.c 中增加 ipm.ipm 函数。

6.4.5 文档

首先查看 PyMite-0.9 中自带的文档，具体参见表 6-4。

表 6-4 PyMite-0.9 文档列表

文 件 名	说 明
AssertStatement.html	关于 Asset 关键字的实现
BuildSystem.html	Make/scons 构建 PyMite
ClassesDesign.html	遗漏的文档：类的字节码实现细节
Closures.html	遗漏的文档：闭包的实现
DevelopmentProcess.html	利用 Subverion、Google code 开发，已经失效
ErrorsAndExceptions.html	错误与异常的特殊实现
FrequentlyAskedQuestions.html	常见问答，包括许可证
GeneratorDesign.html	涉及生成器、迭代器、生成器表达式、协程的实现
HeapDesign.html	关于堆和垃圾回收算法
HowToPortPyMite.html	移植所需条件等
HowToProgramTheAt91sam7s64-ek.html	特定平台下载固件指南

续表

文 件 名	说　　明
HowToReleasePyMite.html	作者本人如何发布 PyMite 的步骤
HowToUsePyMite.html	运行用户模块及脚本的方法
InteractivePyMite.html	使用 IPM 交互模式，即 PyMite 中的 REPL 模式
ModuleImages.html	创建模块，包括标准库模块和用户模块
PyMiteFeatures.html	PyMite 特性表，非常重要，包括关键字、识别符、字符串、布尔量、数值、浮点数、序列类型、字典、操作符、语句、内置函数、系统模块
P14pOverview.html	PyMite/p14p 工程的来历
StringObjects.html	字符串对象操作，声明不支持 UTF-8
Testing.html	介绍了 PyMite 可以使用的测试框架，如 MinUnit 等

文件夹中的文档如果从 index.html 读取，会遗漏若干文档无法查看，所以最好从文件夹中逐个文件查看。这些文档值得读者仔细研读。假设读者已经对 Python 有一定程度的了解，那么阅读这些文档时需要注意与标准桌面版 Python 之间的区别。对于想要移植 PyMite 的用户来说，BuildingSystem/HowToPortPyMite 是必读的两篇文档。

6.4.6　源码树

我们先看看 PyMite/pymbed 的源码树（见图 6-7）。

PyMite 和硬件有关的 makefile 与源码放置在 platform 文件夹中。platform 文件中包含了不同的 MCU 架构，每个 MCU 架构文件夹中主要有：

- main.c，或 main.cpp，视 MCU 与 C/C++编译器平台而定，为系统主循环；
- main.py，启动 PyMite 的 IPM，类似于桌面 Python 的 REPL；
- plat.c（.h），或 plat.cpp，主要是 MCU 平台与 VM 之间的基本接口，如时钟、初始化、存储器读取和写入；
- pmfeature.py（.h），从 Python 常数和宏定义两方面定义 VM 的特性表，用于裁减和扩展；
- makefile，主要的编译控制文件；
- README，与架构有关联的文档。

因架构和编译器不同，有可能有 BSP 和 C 编译器的启动配置文件之类的。例如，mbed 文件夹中还有：

- main_img.c，使用工具将编译后的用户 Python 应用脚本转义为 C 的字节数组，将用户脚本固化在 ROM 中；
- main_nat.c，用户扩展函数的 Native 原生代码，是硬件平台与 VM 之间互联互通的主要途径；

- mbed.py，是 mbed API 映射到 Python 的类库；
- mbed.lib，mbed C++ API 库文件。

```
F:.
├─docs
│  ├─html
│  │  └─dox
│  ├─img
│  └─src
└─src
    ├─lib
    ├─platform
    │  ├─arduino_mega
    │  ├─at91sam7s-ek
    │  ├─avr
    │  ├─desktop
    │  ├─econotag
    │  ├─mbed
    │  ├─mbed2
    │  ├─mosync
    │  ├─pic24
    │  │  ├─common
    │  │  ├─include
    │  │  │  ├─devices
    │  │  │  └─doxygen
    │  │  ├─system_test
    │  │  ├─tools
    │  │  └─unit_test
    │  ├─stm32
    │  ├─teensy
    │  ├─windows
    │  └─_unmaintained
    │      └─mmb103
    ├─tests
    │  ├─system
    │  └─unit
    ├─tools
    └─vm
```

图 6-7　源码树

要获知文件的具体使用目的，最直接的是查看当前文件夹的 makefile。

解释器的核心源码独立于硬件架构，被放置在 VM 文件夹中：

- bytearray.c，bytearray 数据类型；
- class.c，类库实现；

- codeobj.c，用户脚本对象；
- dict.c，python dict 数据类型；
- float.c，浮点数数据类型；
- frame.c，Python 函数所需要的结构体 Frame；
- func.c，函数对象；
- global.c，Python 虚拟机的特性配置；
- heap.c，Python 虚拟机的存储基础堆的管理；
- img.c，用户代码映像以及路径管理；
- int.c，整数类型以及工具方法；
- interp.c，解释器线程以及字节码定义；
- list.c，List 数据类型；
- mem.c，不同存储器工艺（RAM/ROM/NVM）以及对应方法；
- module.c，对于 Python import 的模块支持；
- obj.c，对象管理；
- pm.c，虚拟机的配置；
- pmstdlib_img.c，标准用户脚本转义的 C 字节数组；
- pmstdlib_nat.c，默认的 C/C++/Python 方法；
- seglist.c，用于实现 Python List/Dict 的底层数据结构 Segmented List；
- seq.c，Python iterable 方法；
- sli.c，命令行接口相关方法；
- strobj.c，Python string 数据类型；
- thread.c，Python 线程；
- tuple.c，Python tuple 数据类。

每个 C 源文件都有对应的头文件。此外，plat_interface.h 是用于硬件平台相关接口的头文件，是 platform 文件夹中 plat.c 对应的头文件，放置在 VM 文件夹中。

从源码结构来看 PyMite 基于 Python 2.6，已经开始以 bytearray 字节数组为数据存储主要类型了。和后面要介绍的 MicroPython 相比，PyMite 优先实现了 thread。

在 tools 文件夹中有若干 Python 脚本工具，主要用于 ROM 固件格式转换以及固件合并。

6.4.7　使用流程

假设用户已经将 PyMite 移植到目标处理器上，那么让自己编写的 Python 脚本在 PyMite 上运行的步骤如下。

（1）编写源码：在大多数情况下，用户需要编写 Python 应用脚本，同时还有一个很小的 C 文件，内有系统的 main 函数。

（2）字节码编译及转换：源码写好后，使用 pmImgCreator 工具将 Python 应用转为字节码映像文件，并以字节数组形式形成一个 C 文件。

（3）C 语言编译链接：将转换后的字节数组 C 源码与 PyMite 虚拟机库（libpmvm）一起编译链接为一个 hex/bin 文件。

（4）下载固件：将 hex/bin 下载到目标 MCU 中去。

这意味着，用户每改动一次源码，需要重复一次字节码编译、C 语言编译和烧录系统固件的过程。整体流程显得很烦琐。所以 Okamoto 提供了一个 Google Cloud 服务，让用户可以在线修改源码以直接生成最终固件 HEX 码。这是一个很有想法的设计，但是 Google Cloud 在国内无法访问，必须采用本地 Cloud 或虚拟机来替代。

用户代码：

```
def calc(n):
    return (n-42)

print calc(6*7)
```

注意 PyMite 应用并不像桌面 Python 那样使用** if __name__ == "__main__": **这种语句，不过可以使用 **if ismain():**语句替代。以上代码在桌面版本和 PyMite 虚拟机中均可以运行。

PyMite 虽然做了很多针对嵌入式相关的优化，但和桌面版以及其他版本的嵌入式 Python 相比，其缺乏了最主要的第三方库扩展。这将成为 PyMite 推广普及路上的绊脚石。不过，这依然不妨碍 PyMite 成为特定平台的定制编程环境。

6.4.8 实践

笔者将 PyMite for STM32F103RB/mbed 代码在 mbed 中构建后，以 Keil uVision4 的形式导出工程，并做了一些测试。PyMite 原始设计基于 GCC，之所以导出到 Keil uVision4，是考虑到嵌入式 GCC/GDB 调试的不易，还是 Keil 更加合乎固件开发者的使用习惯。但是笔者发现在仿真器中没有运行任何代码，因为笔者没有将用户 Python 脚本编译进去。

在本书编写之际，笔者总算登录了 pymbed.appspot.com，整个构建过程：编译、下载、运行均正常。Okamoto 在 BitBucket 上提交了其 appspot 的所有源码。有兴趣的读者可以下载其源码，并将其移植到国内的新浪云中，或者移植到更加通用的 Flask/Klein 等小型框架中。

笔者读取其源码后，发现各个硬件平台预先已经有了 bin 文件。而 Web Server Handler 会将用户脚本编译，并与预编译的 bin 文件整合成新的 bin 文件。下面是 Okamoto 提供的 appspot server 源码中实现 bin 文件与用户代码拼接的代码片段。

```
pic.set_options("dummy.img", '.bin', 'usr', 'flash', None, \
    [["main.py", source], "mbed.py"])
pic.convert_files()
logging.info(pic.imgDict["fns"])
img = string.join(pic.imgDict["imgs"], "")
logging.info("target: %s, img size: %d bytes, md5: %s", target, \
    len(img), md5.new(img).hexdigest())
bin_name = target_info[target]['bin_name']
pymite_bin = target_info[target]['pymite_bin']
usr_pos = target_info[target]['usr_pos']
bin = open(pymite_bin, "rb").read()
pad_size = usr_pos - len(bin)
if pad_size < 0:
    logging.error("pad_size: %d", pad)
    self.html_output("error-03")
new_bin = bin + '+'*pad_size + img
logging.info("bin size: %d bytes, md5: %s", len(new_bin), \
    md5.new(new_bin).hexdigest())
self.response.headers['Content-Type'] = "application/octet-stream"
self.response.headers['Content-Disposition'] = 'attachment; filename="%s"' % bin_name
self.response.out.write(new_bin)
```

其中很关键的是这一句：

```
new_bin = bin + '+'*pad_size + img
```

Okamoto 提供的在线开发源码中需要合并两部分代码：用户脚本以及平台解释器。其中，用户脚本编译成 Python 字节码后转换成 C 数组放置在预定义地址中；对于各个平台解释器的 bin 文件，请在 mbed 网站中找到 pymbed 源码，选择对应硬件平台后，在 mbed 在线编译环境中单击编译即可导出 bin 文件。

我们根据 Okamoto 源码，可以做些改进：把 IAP bootloader 整合到平台固件中，Python 代码编译后的字节码 bin 文件通过 bootloader 下载到固定位置，即可以运行。

6.4.9　工程小结

现阶段，PyMite 已经算是比较老的实现版本了。v09 是 Python 2.6 版本，而 pymbed 可以实现 Python 2.7。只有经过合适的适配和产品定制，PyMite 才能够适应商品化开发。开发商可以将 PyMite/pymbed 以 bin 文件格式发布更新，隐藏底层细节；也可以实现自己的安全算法，保护系统源码不被泄露等。而用户代码可以通过在线、离线的手段进行编译、加载、运行。而这这个流程在移动互联时代可以在任意移动终端上实现。

总的来说，PyMite/pymbed 期待有人引领。

6.4.10 网络资源

PyMite 的各类资源分散在不同网站，现汇集如下：

- http://code.google.com/p/python-on-a-chip/
- https://developer.mbed.org/users/dwhall/notebook/python-on-a-chip/
- https://bitbucket.org/va009039/pymbed/
- http://pymbed.appspot.com/
- http://www.youtube.com/watch?v=Oyqc2bFRW9I
- http://pymite.python-hosting.com/
- http://www.deanandara.com/PyMite/2010-State.html
- http://ftp.ntua.gr/mirror/python/pycon/papers/pymite/（有内部结构实现分析）

6.5 VIPER/Zerynth

VIPER（Viper Is Python Embedded in Realtime）是基于 Python 的 IoT 套件。VIPER 是缩写，其英文原意是：毒蛇，蝰蛇。或许开发商觉得这和“蟒蛇”Python 可以拉上表亲关系。:)

VIPER 于 2015 年夏天开始在 Kickstarter 上众筹。其原来的域名（viperize.it）显示出它的母公司 Kinzica Ventures 最初是一家意大利公司，位于比萨，并在美国纽约设立了分公司。VIPER 徽标如图 6-8 所示，后来笔者发现其域名更新为 zerynth.com，而 Zerynth 徽标如图 6-9 所示。原因是 VIPER 商标已经被人注册了，不得不改名。这也导致其所有产品尤其是软件包的包名也做了更改。所以原来安装过 VIPER IDE 的用户，必须重新下载、安装一次 Zerynth Studio。

图 6-8 VIPER 徽标

图 6-9 Zerynth 徽标

Zerynth 是一种易于使用的专业化开发套件，专门针对交互产品、艺术产品和互联网/联网设备的高层设计和跨平台设计。Zerynth 是针对基于 ARM Cortex 32 位微控制器、新型传感器、

执行器和扩展板的交互设备与应用的开发利器。开发者可以利用这些开发板硬件平台进行原型验证，可以利用其规模效应和成熟生产线实现产品化，并快速推向市场。同时，Zerynth 通过各种编程实例和参考设计帮助设计者将原型设计进行快速软件的产品化。

Zerynth 宣称自己是开源设计，这一点笔者曾经有所怀疑。后来发现 Zerynth Studio 底层的 Python 代码都留着，所以其的确是开源的。实际上 Zerynth 安装后，PC 中多了 Python 3.4 与扩展库，以及 Zerynth Studio 和后台工具。这些都可以在安装路径中找到源码，但是需要开发者仔细阅读源码才能够理解其中的奥妙。

6.5.1　硬件平台

Zerynth 可以运行于许多 ARM 微控制器平台，包括 Arduino DUE、UDOO、ST NUCLEO、Spark Core 和 Spark Photon。图 6-10 显示了如何在 Zerynth Studio 中选择硬件。Zerynth 还在不断增加更多硬件。Zerynth 所支持的硬件列表如表 6-5 所示。

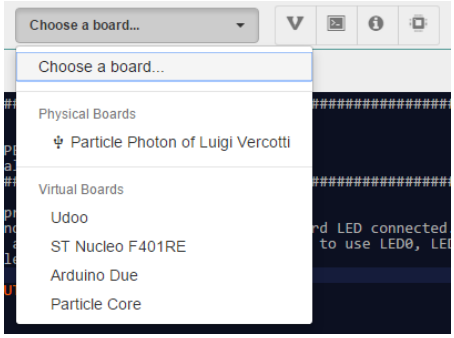


图 6-10　在 Zerynth Studio 中选择硬件平台

表 6-5　Zerynth 所支持的硬件列表

开发板	微控制器	内　　核	RAM（96KB）	ROM（KB）	频率（MHz）
Arduino DUE	Atmel SAM3X8E	Cortex-M3	96	512	84
UDOO	Atmel SAM3X8E + FSL i.MX6	Cortex-M3	96	512	84
ST NUCLEO	STM32F401RE	Cortex-M4	96	512	84
Spark Core	STM32F103CB+TI CC3000	Cortex-M3	20	128 + 2MB SPI Flash	72
Spark Photon	STM32F205RG	Cortex-M3	128	1024	120

其中 UDOO 由 Cortex-M3/A9 两枚处理器构成，M3 中可以运行 Zerynth，而 A9 中可以运行 Android 中的 Zerynth APP。Spark 产品线也是一度在 Kickstarter 上众筹成功的产品，后来改名为 Particle。这可能和 VIPER 的改名原因一样，有太多同类产品重名了。Zerynth 的设计从一开始

就是针对 STM32F103/20X 这些级别更低的 MCU。但是其市场影响力却不如 MicroPython 大。Zerynth 支持 NUCLEO-F401RE 开发板，价钱并不贵，可以在网上直接购买使用。

Zerynth 宣称让开发者关注应用本身，而不必纠缠在没有附加值的底层开发，如针对开发板的 I/O 定制、设备驱动、存储器管理等细节。Zerynth 内置 RTOS（FreeRTOS/ChibiOS）支持。

Zerynth 的组成部分如下：

- Zerynth VM 虚拟机是 Zerynth 核心部分，支持 Python 3 脚本，可以编写不依赖于特定硬件、可重用的高层应用代码。
- Zerynth Studio 是专用开发环境，在 IDE 中开发代码，用户可以在本地和云之间同步代码。同时通过 IDE 可以管理不同硬件平台，并为不同硬件下载不同虚拟机。不过，国内用户更新 IDE 或许会遇到联网问题。
- Zerynth APP 是相关的简单 APP，用户无须编写任何代码就可以控制设备。
- Zerynth Shield 是多传感器扩展板，可以在 Arduino 和 Particle 平台上使用。
- Zerynth Cloud 是其隐含的服务，通过 Git 来配合 Studio 做源码管理等。

Zerynth 提供的 Python IoT 开发环境更加完整，不仅涉及 VM 和嵌入式相关的扩展库，还提供了 IDE 和 APP 开发，鼓励商业化开发。

6.5.2 Zerynth Studio

对于开发者来说，如果有单独的 IDE，可以集约化地满足所有研发需求是一种很好的体验。Zerynth Studio 是采用了浏览器方式设计的 Web GUI。它包括以下特性：

- 代码编辑器支持语法高亮、多标签页、自动填充和错误高亮显示；
- 集成 Python 和 GCC 编译器；
- 模块化设计的代码上传器（uplinker）；
- 电路板发现及管理工具条；
- 集成调试终端和串口监视器，支持检索和自动连接功能；
- 工程数据库支持标签化和搜索功能；
- 支持云同步的本地 Git 存储；
- Zerynth 应用模板 HTML5 编辑器；
- 支持皮肤与主题；
- 自动更新；
- 直接在 GitHub 上提交/分支工程；
- 集成离线文档、电路板特性介绍和引脚信息；
- 集成用户教程；
- 无数的代码示例。

笔者在 Windows 7/Windows 10 以及 Ubuntu 上分别安装过三次 VIPER/Zerynth。安装后的 Zerynth Studio 如图 6-11 所示。应该说 Zerynth Studio 考虑得还是很周到的。而且其设计的架构有些大，集成了 Python/GCC/文档/Git 等。但由于中国的使用环境与国外存在较大区别，因此笔者使用中出现了不少问题。虽然这不是开发团队的责任，却的确是中国境内用户使用中存在的问题。Zerynth 创始人和开发团队成员都非常热情，他们不断通过论坛和电邮努力地帮助笔者解决问题，并通过 VirtualBox 虚拟机提供 Linux 版的 Zerynth Studio 解决 IDE 配置问题。

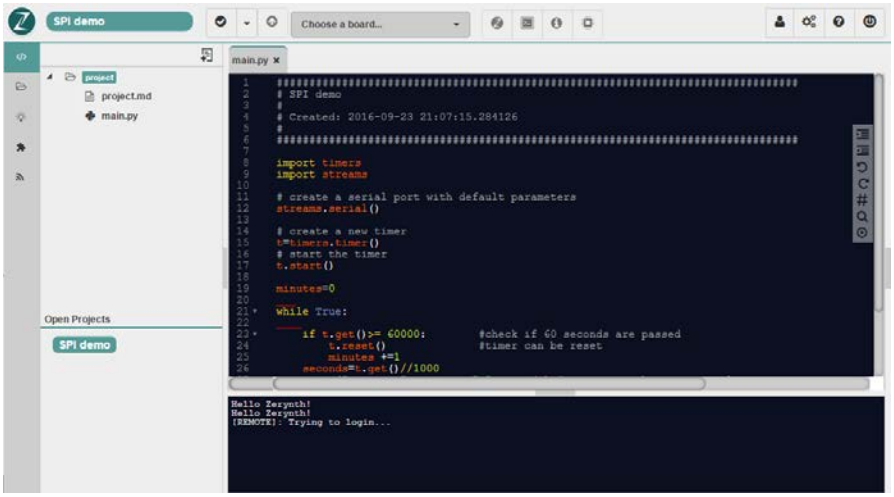


图 6-11　Zerynth Studio

6.5.3　与标准 Python 的区别

Zerynth VM 是针对嵌入式设备而设计的 Python 实现。Zerynth 是 Python 的子集，因为桌面版 Python 的某些特性对于嵌入式编程来说过于浪费资源了。

Zerynth 没有实施的 Python 3.4 特性如下：

- 没有带字符参数的 getattr 和 setattr；
- 不得超过 65535 个名称（对象名）；
- 函数不得超过 256 个参数，没有 **kwargs 支持；
- sequence/map 限制在 65535 个元素；
- 用内置和其他模块方式，而非采用标准 Python 库；
- 没有闭包、生成器、装饰器；
- 没有 eval()和 compile()。

Zerynth 的附加特性如下：

- 内置实时线程；

- 内置原生 C 函数调用；
- 异常不是类，而是带有可选错误码的名称；
- 支持与硬件相关的内置类，如 GPIO、ADC、PWM 等。

Zerynth Studio 很好用，它和 ARM mbed 有着异曲同工之妙。在安装 Zerynth Studio 的过程中，发现它本质上只是一个安装器，其还会不断地从云服务器中下载更新各类组件，包括 arm-gcc、runtime、lib 等。这本来是为了方便开发者的做法，但是由于其组件大量使用的都是国外的网络服务，比如 Google、GitHub、AWS 等，因此在当前的国内联网环境中却给大家造成了很大的困扰。其主要问题是连接速度特别慢，甚至堵塞，下载的组件也可能不完整，以致会遇到各种古怪问题。而且，短期内该问题还无法得到解决。

Zerynth Studio 实际上是 Node.js 中 WebKit 包和 Python 整合的产品，看上去很炫酷。其实笔者觉得 GUI 采用 wxPython 运行更加稳定。

6.5.4 快速启动

具体步骤如下：

（1）下载并安装 Zerynth Studio。其有 Windows/Linux/Mac 三个版本，都是 64 位版本。

（2）创建 Zerynth 用户名，并使用电邮注册账号。该账号主要用于技术支持、工程的云端备份和同步，获取自动更新等。

（3）一旦验证通过，使用账号自动登录 Zerynth 云服务。

（4）为了防止被覆盖，Zerynth 的例程不能够直接编辑。开发者必须复制（Clone）例程，然后修改代码。

（5）无论是复制代码还是创建工程，都可以自定义标签。Zerynth 通过这些标签来简化管理和复用工程代码。

（6）连接电路板，可以将其重新命名以便日后识别。

（7）选中该电路板，必须再将这些电路板 Viperize 化（现在改名为 Virtualized/虚拟化）。所谓虚拟化就是将 VIPER VM 安装在电路板的过程。在虚拟化过程中，会针对电路板下载特定版本的 Zerynth VM。

（8）编译用户脚本（Zerynth 称之为 Verify）。

（9）复位电路板，Zerynth Studio 会尝试与 Zerynth VM 通信，并上传（Uplink）到电路板。

（10）如果一切顺利，现在就可以运行了。

6.5.5 坎坷的使用过程

Zerynth 使用的核心步骤可以简化为以下三步。

（1）Viperize：将 Zerynth VM 下载到目标 MCU，即虚拟化；

（2）Verify：编译用户 Python 脚本；

（3）Uplink：将编译后的 Python 脚本上传到目标 MCU 的用户空间。

1．虚拟化

在 Zerynth 虚拟化的过程中有失败的可能。即由于某些原因无法将 VM.bin 下载到 NUCLEO-F401RE 中去。如果虚拟化失败，可以在 Viperize 过程中保存对应的 bin 文件，再通过 NUCELO 开发板板载调试器的 MSD Bootloader，将 bin 文件复制下载到目标 F401RE 中去。

2．编译代码

在 Linux 版本中，可能会遇到 Git 所导致的无法 clone、fork 和 publish 代码等问题。必要时可以安装 Git 客户端。

3．上传代码

在笔者的实验过程中，上传用户代码（uplink）遇到了一些困难。此时要确保 uplink 编辑后，需要按下已经虚拟化的 F401RE 的复位按键，才能够进入 uplink 流程。笔者在 Windows 7/Windows 10/Ubuntu 16.04 中安装 Zerynth 时遇到了不少困难，最终 Zerynth 公司协助笔者将问题定位到了 STLINK/V2 的固件版本上。

这说明，Zerynth 下载 VM 后，在 Zerynth Studio uplink 流程中，USB 串口通信、IAP 协议、Flash 烧录、校验、运行的整个链条还需要持续优化。

6.5.6　Zerynth 目录结构

Zerynth 团队与用户群都为 Zerynth 社区提供了大量有意义的例子和扩展库。和 PyPI 方式不同，其采用的是类似于 mbed/GitHub 的社交化源码管理。我们先看看 Zerynth 提供的库。查看库可以从 Studio 中查看，即 Studio 中的“Package Installer | Package Manger | All Packages”，在此可查看到所有的软件包。假设其安装路径为 C:\Users\user_name\zerynth，可以在其子目录中查看各个包的源码（见表 6-6）。

表 6-6　Zerynth 包

分　类	中　文	内　　容	子目录
Libraries	库	Wi-Fi 及传感器设备驱动库	env\libs\official
VM	虚拟机	开发板的虚拟机	env\core\official\vm
Boards	BSP	编译器、下载器、VHAL 等常用工具	env\core\official\boards
Core	内核	编译器、文档、例程、Studio、标准库、工具箱、uplinker、虚拟机和包管理器	env\core
Sys	系统	GCC、下载器、浏览器、DFU、Git、运行时	env\sys

续表

分　类	中　文	内　　容	子目录
VHAL	硬件抽象	编译所需的头文件	env\core\official\vhal
examples	例程	例子程序、演示代码	env\core\official\examples
stdlib	标准库	标准库模块	env\core\official\stdlib

可以说，除了 Zerynth 的 VM 源码没有提供，其他的都提供了。

6.5.7　硬件相关库

如图 6-12 所示，单击 Zerynth Studio 左侧的小灯泡图标就可以看到各种例程，用户可以从中复制（clone）代码，编译并上传到目标板中去。

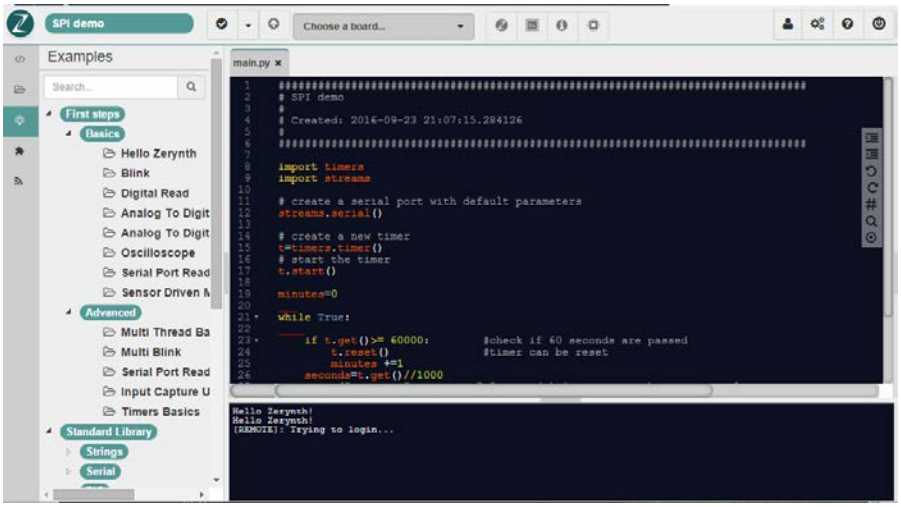


图 6-12　Zerynth 内置例程

6.5.7.1　ADC 采样

ADC_Acquistion.py：

```
import streams  # import the streams module
import adc      # import the adc driver

# create a stream linked to the default serial port
streams.serial()

while True:
```

```
# Basic usage of ADC for acquiring the analog signal from a pin
    value = adc.read(A0)
    print("One sample:",value)

# The complete definition of adc.read() is adc.read(pin, samples=1)
# For an advanced usage of adc.read refer to the official ZERYNTH documentation

#acquire 10 samples with default sampling period
    value2 = adc.read(A0,10)
    print("10 samples:\n",value2)

#acquire 3 samples from the first 4 analog pins with default sampling period
    value3= adc.read([A0,A1,A2,A3],3)
    print("3 samples from A0, A1, A2 and A3:\n",value3)

    print()
    sleep(300)
```

其 adc.read 可以读取单一引脚，也可以读取选中的一组引脚，还可以设置采样数量。

6.5.7.2　多线程驱动 LED

MultiBlink.py：

```
# Initialize the digital pins where the LEDs are connected as output
pinMode(D2,OUTPUT)
pinMode(D8,OUTPUT)
pinMode(D5,OUTPUT)

# Define the 'blink' function to be used by the threads
def blink(pin,timeON=100,timeOFF=100):
    while True:
        digitalWrite(pin,HIGH)   # turn the LED ON by making the voltage HIGH
        sleep(timeON)           # wait for timeON
        digitalWrite(pin,LOW)    # turn the LED OFF by making the voltage LOW
        sleep(timeOFF)          # wait for timeOFF

# Create three threads that execute instances of the 'blink' function.
thread(blink,D2)
# D2 is ON for 100 ms and OFF for 100 ms, the default values of delayON an delayOFF
thread(blink,D8,200)
# D8 is ON for 200 ms and OFF for 100 ms, the default value of delayOFF
thread(blink,D5,500,200)     # D5 is ON for 500 ms and OFF for 200 ms
```

在以上例子中，Zerynth 已经实现了线程。

6.5.7.3　串口的多种读取方式

SerialPort.py：

```
import streams

s = streams.serial()

# Testing various serial port reading methods

while True:
    print("write some chars and send it to the board")
    char=s.read()
    # read and return any single character available on the serial port one by one
    print("This is the first char you wrote:",char)
    print() # add a line space for improving the serial console ouput readability

    sleep(500) # waiting for the serial buffer to fill
    length=s.available() # check if data are available on the port and count them
    chars=s.read(length)
    # read all the bytes available in the buffer an return the bytearray
    print("This are the other", length, "chars you wrote:",chars)
    print() # add a line space for improving the serial console view

    print("write a line ending it with return or enter")
    line=s.readline()
    # read until a line terminator \n is found, then return a bytearray
    print("This is the line you wrote:",line)
```

Zerynth 将串口作为流式接口的类。Zerynth 提供了从串口读取单一字符串、读取规定长度字符串以及读取整行字符串的不同方法。

6.5.7.4 PWM 捕捉单元

InputCaptureUnit.py：

```
import pwm
import icu
import streams

# create the serial port using default parameters
streams.serial()

# define a pin where a button is connected, you can use the Nucleo button pin \
# as input or change it with any other digital pin available

# this is the pin the button is connected to; for the various supported boards, \
# ZERYNTH automatically translates it
# to the board button. On Arduino DUE the user button isn't installed, \
# so this line rises a compilation error.
# On Arduino DUE, change it to the pin your button is connected to \
# or comment this line (see below)
```

```
buttonPin=BTN0


# define the ICU pin. D5 works with Arduino footprint boards \
# while with Particle boards D0 can be used
captPin=D5.ICU  # On Arduino like boards
#captPin=D0.ICU  # On Particle boards

# define the PWM pin. D13 works with Arduino footprint boards \
# (and is also connected to a LED) while with Particle boards A4 can be used
pwmPin=D13.PWM # On Arduino like boards
#pwmPin=A4.PWM # On Particle boards

# set the pin as input with PullUp, the button will be connected to ground
pinMode(buttonPin, INPUT_PULLUP)

# define a function for printing capture results on the serial port
def print_results(y):
    print("Time ON is:", y[0],"micros")
    print("Time OFF is:",y[1],"micros")
    print("Period is:", y[0]+y[1], "micros")
    print()

# define a global variable for PWM duty cycle and turn on the PWM

duty=10
pwm.write(pwmPin,100, duty,MICROS) #pwm.write needs (pn, period, duty, time_unit)

# define the function to be called for changing the PWM duty \
# when the button is pressed
def pwm_control():
    global duty
    duty= duty+10
    if duty>=100:
        duty=0
    pwm.write(pwmPin, 100, duty,MICROS)
    print("Duty:", duty, "millis")

# Attach an interrupt on the button pin waiting for \
# signal going from high to low when the button is pressed.
# pwm_control will be called when the interrupt is fired.
# If you are on Arduino DUE and you haven't connected any button
# comment the following line
# you will not change the PWM duty but you can still test the ICU capture
onPinFall(buttonPin, pwm_control)

while True:
    # start an icu capture to be triggered when the pin rise.
```

```
    # this routine acquires 10 steps (HIGH or LOW states) or
    # terminates after 50000 micros
    # this is a blocking function
    x = icu.capture(captPin,LOW_TO_HIGH,10,50000,MICROS)
    print("captured")
    # x is a list of step durations in microseconds,
    # pass it to the printing function and check the serial console
    print_results(x)

    sleep(1000)
```

PWM 的用途很多，该段代码提供了 PWM 捕捉与输出等实例。定时的捕捉还可以用于红外和其他调制解调技术。

6.5.7.5 红外解码

InfraRed_Raw_Capture.py：

```
import icu
import streams
import pwm

streams.serial()

# Set the pin the IR receiver is connected to.
# In this example D2 is used:
# you are good to go.
ir_pin = D2.ICU


def IR_capture():
    while True:
        print("Capturing...")
        # Starts capturing from the icu configured pin.
        # The capture starts from a selected trigger
        # (in this case capture will start when the pin first goes from
        # HIGH to LOW).
        # The max number of samples to be collected and
        # a maximum time window are specified.

        # Play with max number and time window to fit your remote protocol.
        # The following values are for the NEC IR (used by LG) protocol
        x = icu.capture(ir_pin,LOW,67,68,pull=HIGH)
        print(x,"\n captured n samples:",len(x))


# captures in a different thread
thread(IR_capture)
```

```
while True:
    print("alive!")
    sleep(1000)
```

确切地说，这段代码还不完整，只是捕捉了原始码流，还需要进一步解码。

6.5.7.6　DAC 数模转换

DAC_Basic.py：

```
import streams
import adc # for analogRead

import dac


def readInput():
    while True:
        print("reading A1: ",analogRead(A1))
        sleep(500)

streams.serial()
pinMode(A1,INPUT)

# read input in a separate thread
thread(readInput)

my_dac = dac.DAC(D8.DAC)
my_dac.start()

# circular mode: continuously repeat the input buffer
my_dac.write([100,200,900,800],1000,MILLIS,circular=True)
```

dac.write 支持无限循环写法。这种编程可以大大简化应用程序的编写。

6.5.7.7　多线程

Thread_Producer_Consumer.py：

```
import threading
import streams
import queue


streams.serial()

# create a bounded queue
q = queue.Queue(maxsize=20)
```

```
# keep producing an element every 100 millis
def producer(id):
    while True:
        try:
            x = random(0,100)
            print("producer",id,"->",x)
            q.put(x)
        except Exception as e:
            print(e)
        sleep(100)

# keep consuming an element every 1 second
def consumer(id):
    while True:
        try:
            print("consumer",id,"<-",q.get())
        except Exception as e:
            print(e)
        sleep(1000)

# start everyone
thread(producer,0)
thread(consumer,1)
thread(consumer,2)

while True:
    isfull = q.full()
    print("Queue is full?",isfull)
    if isfull:
        # clear queue if full
        print("Clearing queue")
        q.clear()
    sleep(5000)
```

此例是经典的生产/消费模式，利用多线程和队列数据结构，可以实现较为复杂的消息队列和任务调度。

6.5.7.8 按键中断程序

interrupt.py：

```
import streams

# create a serial port stream with default parameters
streams.serial()

# define where the button and the LED are connected
```

```
# in this case BTN0 will be automatically configured
# according to the selected board button
# change this definition to connect external buttons on any other digital pin
buttonPin=BTN0
ledPin=LED0  # LED0 will be configured to the selected board led

# configure the pin behaviour to drive the LED and to read from the button
pinMode(buttonPin,INPUT_PULLUP)
pinMode(ledPin,OUTPUT)

# define the function to be called when the button is pressed
def pressed():
        print("touched!")
        digitalWrite(ledPin,HIGH) # just blink the LED for 100 millisec
        sleep(100)
        digitalWrite(ledPin,LOW)

# attach an interrupt on the button pin and call the pressed function when it falls
# being BTN0 configured as pullup, when the button is pressed
# the signal goes to from HIGH to LOW.
# opposite behaviour can be obtained
# with the equivalent "rise" interrupt function: onPinRise(pin,fun)
# hint: onPinFall and onPinRise can be used together on the same pin,
# even with different functions
onPinFall(buttonPin,pressed)
```

作为嵌入式系统中最常见的按键处理，采用了中断驱动方式。

6.5.7.9　异常处理

exception_handler.py：

```
import streams

streams.serial()

while True:
    for x in range(-10,10,1):   # create a loop ranging between -10 and 10

        try:                   # open the Exception monitoring scope
            value=100//x        # when x=0 this will results in a DivisionByZero!
            print(value)
        except Exception as e:  # capture any raised exception as e
            print(e)            # print the content of e to monitor
                               # click on the console X icon to open debugger window
        sleep(1000)
```

以上程序运行时会抛出 divide by zero 异常。Zerynth Studio 有一个独特的 Debugger，可以

捕捉到此类异常。

当检测到异常发生时，该信息会通过终端打印出来。该终端通过特殊的错误图标将异常表示出来以区别于其他打印信息。单击该图标后，可以浏览异常的相关信息，如图 6-13 所示，显示异常的回溯堆栈 traceback（最多可以显示四层调用关系）。

图 6-13　Zerynth 异常浏览器

如果使用单独的串口来打印异常，则可以通过“@”符号后的“异常坐标”来获取 traceback 信息。此外，如果开发者对于底层很了解的话，从调试窗口就可以很容易地检查 Zerynth 字节码。

6.5.7.10　资源文件

resource.py：

```
import streams

streams.serial()

# new_resource is a builtin that includes the file specified
# in the generated bytecode
# remember: the file used as a resource must be added to the project!
new_resource("mytxt.txt")

# to open resources saved by new_resource
# a specific url must be passed to open
# once opened, methods like read, readline and seek can be used!
ff = open("resource://mytxt.txt")

print("Opening resource...")
while True:

    # let's read line by line and print
    line = ff.readline()

    # if line is empty, we are at the end of file
    if not line:
        break

    print("Line:",line)
```

```
# easy!
print("Done!")
```

工程中所需的资源文件，如文本或其他二进制资源文件（如图形、声音等），都可以编译成 bytecode，并通过 resource 句柄方式传递给外部客户端。

6.5.7.11　Flash ROM 访问

flash_internal.py：

```
import streams
import json
import flash

## Warning! This example works on the Particle Photon only!
# You need to change the flash address to use another board
#  --> For Sam3X based boards you can safely use 0xe0000

streams.serial()


print("create flash file")

# open a 512 bytes FlashFileStream at address 0x80E0000
ff = flash.FlashFileStream(0x80E0000,512)

print("reading flash file")

for i in range(30):
    print(i,"->",str(ff[i]))

print("writing flash file")

sleep(1000)

hh = {
    "type":"thing",
    "data":23.5
}

ds = json.dumps(hh)

# save length and json to flash
ff.write(len(ds))
ff.write(ds)

ff.flush()
```

```
ff.seek(0,streams.SEEK_SET)

print("reading flash file")

n = ff.read_int()

for i in range(n+4):
    print(i,"->",str(ff[i]))
```

将内部 Flash ROM 作为一种数据存储器使用，这个特性比较少见，但是很有用。使用时需要注意，这是一个比较容易出错的环节。Flash 更新需要深入了解 MCU 的空间地址和更新流程。

6.5.7.12 软启动

soft_reset.py：

```
# import the mcu module
import mcu
import streams

# open the default serial port, the output will be visible in the serial console
streams.serial()

resetting = False
# define a simple function to be called on interrupt
def reset():
    global resetting
    resetting=True

# on button pressed, call reset
# >>>> if the board hasn't a button, change BTN0 to a digital pin
#      and use a jumper wire to simulate a falling edge <<<<
onPinFall(BTN0,reset)

# loop forever
while True:
    print("Hello ZERYNTH!")
    sleep(1000)
    # check for the need to reset
    if resetting:
        print("Resetting in 3 seconds!!")
        sleep(3000)
        mcu.reset()
        # bye!
```

许多情况下，需要 MCU 自行软启动，然后进入某些特殊状态，比如 Bootloader 或者 ISP/IAP 模式。但是某些 MCU 软启动会有问题，必要时可以通过外部看门狗来进行硬启动。

6.5.7.13　调用 C 函数

C_Calling.py：

```
import streams

streams.serial()

# define a Python function decorated with c_native.
# This function has no body and will instead call
# the C function specified in the decorator.
# The source file(s) where to find the C function must be given (cdiv.c)
@c_native("_my_c_function",["cdiv.c"],[])
def c_division(a,b):
    pass


while True:
    a = random(0,100)
    b = random(0,10)
    try:
        # call the c_division function with random values
        c = c_division(a,b)
        print(a,"/",b,"=",c)
    except Exception as e:
        print(e)
    sleep(1000)
```

Zerynth 采用装饰器来实现调用原生 C 函数。Zerynth 会使用 GCC 编译 C 语言，并与 Python 代码一起运行。

6.5.7.14　网络服务

Zerynth 内置库中支持 CC3000 和 Boardcom 的 BCM43362 Wi-Fi 芯片。基于这两枚芯片，Zerynth 提供了一些与物联网有关联的例程和库。包括：

- Mini Web Server，Helloworld 网页。
- UDP Pinger，采用收发两个线程实现对于远程服务器的 Ping。
- Wi-Fi Scanner，检索周围的 Wi-Fi BSSID，并打印相关信息。
- HTTP Time，从远程 HTTP Web 服务中获取时间信息。
- HTTP Weather，从 openweathermap.org 服务获取天气信息。

6.5.8　其他特性

在 Zerynth Studio 的线下文档和网站的线上文档中，都是用 reST 和 Sphinx 构建的文档。通

过阅读文档，可以了解 Zerynth 方方面面的知识。笔者个人推荐阅读以下几部分：

- standard libraries（标准库）。包括 Python 3 基础、序列与映射类型、引脚定义、内置类型、异常等。其标准库除了 Threading、stream，还支持 SoftwareTimer、MsgPack、Fifo、Queue 和 Flash 模块。
- virtual machine（虚拟机）。主要包括操作系统抽象层、硬件抽象层和虚拟机接口，这对虚拟机移植很有意义。
- zpm（Zerynth package management），包管理器。基于 Git 的包管理器与 PyPI 有所不同，也要求读者有 Git 的使用经验。
- boards。有关于特定 PCBA 和 MCU 的 Flash 划分，DFU 以及 uplink 的流程在该文档中得到了描述。
- communities and official libraries（社区和官方库）。在开发项目之前，先找找“轮子”。这非常符合 Python 的风格。

6.6 MicroPython

MicroPython（见图 6-14）是 Kickstarter 上众筹成功的众多嵌入式开发板之一。由剑桥大学理论物理学家 Damien George 设计，以 MIT 许可证形式发布，允许商业化。其官网是 micropython.org。

图 6-14 MicroPython pyboard

6.6.1 工程背景知识

MicroPython 实现了完整的 Python 3.4 语法（包括 exception、with、yield from 等），并提供核心的数据类型：string（包括 unicode）、bytes、bytearray、tuple、list、dict、set、fronzenset、array.array、collections.namedtuple，以及类和实例。内置模块包括 sys、time 和 struct。请注意，作为嵌入式 Python，MicroPython 仅仅支持 Python 3.4 的数据类型和模块子集，而非完整特性。

其特性如下：

- 兼容 Python 3 语法。
- 完整的 Python 词法分析器、解析器、编译器、虚拟机和运行时。
- 包括命令行接口，可离线运行。
- Python 字节码由内置虚拟机编译运行（这一点与 PyMite 不同）。
- 高效内部存储算法，带来高效内存利用率。
- Python 装饰器可以用于编译原生代码，带来更快执行速度。
- 使用数值时，可以设置使用底层整数代替 Python 内置数值对象，所以其代码效率媲美于 C，并可以被 Python 调用。其适合时间紧迫性、运算密集型应用。
- 其内联汇编，使得应用可以接入底层，可以像调用 Python 函数一样调用内联汇编器。
- 基于标记的内存垃圾回收算法，运行周期少于 4ms。许多函数可以不使用栈内存段，也可以回避垃圾回收。

在 Python 中使用内联汇编，降低了 MicroPython 的移植性能，但这却是解决某些棘手问题如中断处理 ISR 的必要手段。目前其提供的是 ARM V7-M 的 Thumb2 指令集的汇编支持。这意味着在其他平台甚至是较旧的 ARM7TDMI 上都会存在一定的麻烦。不过 MicroPython 既然可以在 ESP8266 和 PIC16 中得到移植，那么这个问题还是可以解决的。

与 PyMite 相比，MicroPython 支持最新的 Python 3.4 语法，针对嵌入式系统做了不少优化，其是最新的嵌入式 Python 虚拟机。相比之下，PyMite 专注于小型化，MicroPython 则关注用户生态的建设与扩展性。

MicroPython 官方推广的 pyboard 的硬件特性如下：

- STM32F405RG MCU；
- 168MHz Cortex-M4，带 DSP 指令和 32 位 FPU（浮点计算单元）；
- 内置 1MB 闪存，192KB RAM；
- 30 组 GPIO，其中 28 组 5V 兼容；
- 2 组 SPI，2 组 CAN 总线，2 组 I2C；
- 14 组 12 位 ADC；
- 2 组 DAC；
- USB-OTG，支持 CDC、MSD、HID 设备主机和设备；
- LED x 4，复位和用户按键；
- MMA7660，3 轴加速度计；
- SD/TF 卡槽；
- 板载 3.3V 300mA LDO，由 USB 和外置电池供电；
- 实时时钟。

STM32F405RG 还支持 I2S+Audio PLL 音频输入/输出，但这不是 MicroPython 支持的重点，

在其标准库中并没有提供相应的支持。但是已经有开发者正在增加对于 I2S 的支持。最近 MicroPython 正在规划全新的硬件 API 设计，所以需要读者经常性访问其 GitHub 上的更新。

6.6.2 在线评估网页

由于 pyboard 的价格较高，因此笔者为那些手头没有开发板的开发者也提供了开发手段。MicroPython 官网有个 pyboard 在线测试网页，可用于在没有硬件情况下进行远程评估，网址：http://micropython.org/live/。

该网页内除了一块 pyboard，还连接着其他外设：LCD、电机、LED 条等。在网页中有两种输出方式：

- 一种为远程摄像头，可以观察液晶、LED 和电机的变化。
- 另外一种为标准输出，可以观察打印口输出。

用户可以在线编辑代码，在连线的 pyboard 中运行，并通过摄像头和标准输出查看最后的结果。

在该网页中也有各种程序例子。可以使用“EXAMPLE”的左右方向按钮切换不同例程，并可以直接运行。也可以在这些例子上修改后运行。这是没有 pyboard 硬件的情况下运行、测试代码的方式之一。

需要注意的是，由于可能同时有多个人访问这个仅有的 pyboard 实例，所以加载的程序会放在队列中等待运行。在程序运行时，如果有此类情况发生，网页会有提示。

6.6.3 官方硬件平台分支

所谓官方分支是指 GitHub 中的 MicroPython 源码分支。目前已经支持的硬件平台如下：

- STM32F405，pyboard 最初版本，简称 PYB；
- bare-arm，ARM MCU 的裸机最小版本，用于控制代码大小；
- teensy，在 Teensy 3.1 上的版本，使用 Freescale MK20DX256、Cortex-M4 控制器；
- pic16，在 PIC16 上运行的版本；
- cc3200，在 TI CC3200 Wi-Fi SoC 上的版本；
- esp8266，在 ESP8266 上的实验性版本；
- stm32hal，基于 STM32Cube HAL 的版本；
- UNIX，主机上的实现版本；
- minimal，MicroPython 最小集，作为移植到其他 MCU 的起点。

虽然 MicroPython 最初是针对嵌入式系统的，但是作为最活跃的 Python 项目之一，其出现了 UNIX 和嵌入式两种主流分支。而针对两者的库也出现了差异。UNIX 分支可以运行在 UNIX/

Linux 之上，以及嵌入式 Linux 上，其有类似于 pip 的管理组件 upip；而嵌入式分支需要根据硬件来修改 Python 库和定制底层固件。

6.6.4　衍生项目

MicroPython 设计基于 Linux/UNIX 平台，许多开发工具都是基于 Linux 的，当然现在其也提供 Windows 版本。其源码和 EDA 文件在 GitHub 上分享。除了 MicroPython 官方分支，还有另外的衍生项目。

在 GitHub 上，有 224 个软件库与 MicroPython 有关，而 MicroPython 本身有 587 个分支。这远远超越其他版本的嵌入式 Python。在本书编写之际，MicroPython 官网介绍了以下几个版本：

- UNIX，可以运行于各类 Linux/UNIX 系统；
- pyboard，即官方开发板版本，采用 Cortex-M4 内核的 STM32F405RG；
- WiPy，使用 TI CC3200 内置 Cortex-M4，由 PyCom 推出；
- ESP8266，非常流行的国产 Wi-Fi 芯片，没有采用 ARM 内核，而采用了 Cadence 可定制内核 Xtensa LX106。

MicroPython 是比较活跃的嵌入式 Python 项目。在本书编写之际，笔者特意登录了 Kickstarter 网站，发现同时有两个项目在众筹。主要是在原版 pyboard 基础上增加了 BLE/Wi-Fi/LoRaWAN/WSN 之类的选项。经过检索，第三方分支如下：

- OpenMV，基于 STM32F407 + OV9650，可以实现网络摄像头；
- BBC microbit，基于 nRF51822 内置 Cortex-M0 分支；
- WiPy，基于 CC3200 内置 Cortex-M4 内核分支；
- LoPy，具备独立 IDE 的 Python Nano 网关。

其中 LoPy 支持三种无线接入方式，还有多种其他功能：

- Semtech SX1272，可做 LoRa 网关；
- 最新 ESP32，双核 Xtensa 处理，支持 Wi-Fi 和双模蓝牙；
- 专用 PyMakr IDE；
- 支持多个云平台（Bluemix、Azure 及私有云平台）。

WiPy/LoPy 和廉价 Wi-Fi 芯片 ESP8266 分支值得注意。MicroPython 直接运行于 MODEM 的 ROM/RAM 中，而非运行在外部主控 MCU 中。与此同时，George 与 BBC、Python 基金会合作，推出了 BLE 芯片 nRF51822 的 microbit 分支。这说明，MicroPython 的各个分支开始直接进驻智能 MODEM 芯片。这种实施模式非常类似于 Telit MODEM 中的 Python。

原版 MicroPython pyboard 的售价为 20 美元，价格稍贵。不过，开源产品允许复制。开发者除了最初的 STM32F405 之外，还将 MicroPython 移植到了性能稍弱的 F401 和 F411 中。国内

EEWorld 论坛开展过国产 MicroPython 的评估活动。除了 EEWorld 的版本，还有一些类似的 PCBA 如 pyMagic，变化的主要是 I/O 接口形式。笔者计划配合相关论坛继续推出自己的 MicroPython PCBA 分支，主要是针对细分市场的用户需求，支持更多的扩展板，增加更多功能。

6.6.5 UNIX 版本

MicroPython 有 UNIX 版本，在常见的 Linux 如 Ubuntu/Arch/Debian 上一样可以使用。可以在没有硬件的情况下评估 MicroPython。

```
$ mkdir micropython
$ cd micropython
$ wget -c https://github.com/micropython/micropython/archive/master.zip
$ unzip ./micropython.zip
$ sudo apt-get install libffi-dev pkg-config
$ cd unix
$ make
```

以上是在 Ubuntu 中创建文件夹、下载源码、解压缩，安装软件依赖项（Foreign function interface 和 Linux 编译辅助工具）并顺利构建 MicroPython 的过程。完成后可以查看 MicroPython 的帮助信息：

```
$ ./micropython --help
usage: ./micropython [<opts>] [-X <implopt>] [-c <command>] [<filename>]
Options:
-v : verbose (trace various operations); can be multiple
-O[N] : apply bytecode optimizations of level N

Implementation specific options:
  compile-only                -- parse and compile only
  emit={bytecode,native,viper} -- set the default code emitter
  heapsize=<n> -- set the heap size for the GC (default 1048576)
```

用户可以从命令行配置 GC 的堆的大小。1MB 对于 MCU 有些大，但是对于 MPU，即使是 Linux 最小系统也不算特别大。

6.6.5.1 micropython-pip

Python 的普及与功能丰富的标准库及第三方扩展库有密切关联。pip 是桌面 Python 软件包管理工具，MicroPython 也有对应的版本：micropython-pip，简称为 upip。这使得 MicroPython 可以形成一个独立完整的嵌入式 Python 生态。

6.6.5.2 micropython-lib

MicroPython 的作者还维护着 upip 的标准库。micropython-lib 虽然是个实验性的工程，但却

规划得非常完整。

micropython-lib 网址：https://github.com/micropython/micropython-lib。

micropython-lib 的每个模块或者包都可以单独从 PyPI 中安装，每个模块要么重新编写，要么移植自 CPython。micropython-lib 的开发目的主要是为了满足 MicroPython UNIX 版的需求。但实际上每个模块的系统需求是不一样的。只要某个模块与硬件 I/O 无关，该模块应该就可以在 MicroPython 上工作。

换而言之，MicroPython 工程中实现的标准库是最小集合，而剩下的可以在 micropython-lib 中选择。其不仅仅可以用于 UNIX，也可以用于 pyboard 等开发板。

```
$ pip-micropython install micropython-copy
$ micropython
>>> import copy
>>> copy.copy([1, 2, 3])
[1, 2, 3]
```

细心的读者会在下面看到 upip 的另外一种早期用法：

```
$ ./micropython -m upip install micropython-pystone
```

这说明，MicroPython/micropython-lib 的迭代速度非常快。实际上 micropython-lib 已经有多人维护。不过里面的许多库是 dummy，等待有人去“填坑”。读者需要经常去查看这两个工程的进展。如果可以，也可以一起参与“填坑”。这里面有些库还很有实用价值，比如 umqtt，这是一个 MQTT 客户端。

6.6.5.3　UNIX 版本性能测试

我们可以进入 MicroPython 交互终端做一些简单测试。测试完毕后按 CTRL+D 组合键退出终端。

```
$ ./micropython
>>> list(5 * x + y for x in range(10) for y in [4, 2, 1])
[4, 2, 1, 9, 7, 6, 14, 12, 11, 19, 17, 16, 24, 22, 21, 29, 27, 26, 34, 32, 31, 39, 37,
 36, 44, 42, 41, 49, 47, 46]
CTRL+D
```

MicroPython 自身经过大量严密的测试。

```
$ make test
```

我们通过 upip 安装 pystone 软件包对 MicroPython 本身进行性能测试。pystone 是 Python 虚拟机自带的测试组件，也是开源的 Python 脚本，在其他桌面或者嵌入式 python 中也可以实现。不同的实现如 CPython、Jython、IronPython 和 MicroPython 基本上都有这个测试组件。

```
$ ./micropython -m upip install micropython-pystone
```

```
$ ./micropython -m pystone
Pystone(1.2) time for 50000 passes = 0.71
This machine benchmarks at 70422.5 pystones/second
```

笔者比较感兴趣的是 MicroPython 与桌面 Python 的性能比较。于是在同一机器中，做了以下对比。

VirtualBox 中 Ubuntu 12.04 32 位、CPython 2.7 pystone 跑分：

```
$  cd /usr/lib/python2.7/test
$  ./pystone.py
Pystone(1.1) time for 50000 passes = 0.49
This machine benchmarks at 102041 pystones/second
```

原生 Windows 7 64 位、CPython 2.7 pystone 跑分：

```
C:\Python27\Lib\test>pystone.py
Pystone(1.1) time for 50000 passes = 0.591608
This machine benchmarks at 84515.4 pystones/second
```

经过多次测量。VirtualBox 虚拟机中的 Linux/CPython 最快，而 Windows 中的 CPython 和虚拟机中的 MicroPython 虚拟机速度接近。除了 UNIX 版本，MicroPython 的 Windows 分支也已经出现在 GitHub 中了。笔者决定以后再对比一下 MicroPython 和 CPython 在原生 Windows 中的表现。根据 George 的实验，pyboard（STM32F405）上运行的 MicroPython 性能与 AVR 原生代码执行速度相近。

6.6.5.4 UNIX 版本的意义

MicroPython 开发 UNIX 版本有特定的存在意义：

- 无须硬件也可以开发高层算法，开发完毕后与 pyb 包整合即可完成大体设计；
- 软件开发团队协作开发更加容易，避免硬件差异性对于工程进度的影响；
- UNIX 版本为 Linux 最小系统提供了一个标准 CPython 替代品；
- upip 软件管理器有利于 MicroPython 形成自己的生态。

最后两点对于一些工程的平台选择很重要。ARM 平台的更新速度很快，ARM7TDMI 已经被 Cortex-M0/M3/M4 所取代，其已基本进入停产计划；ARM9/ARM11 有不少 MCU/MPU 也进入了清库存的行列。现阶段会出现 ARM9/ARM11 SoM（含 MCU/DRAM/Flash）产品线比新款 M4/M7 MCU 要便宜的情况。典型的例子如下：

- Adafruit 推出的树莓派 Zero（ARM11）促销价是 5 美元；
- FriendlyARM 推出的 NanoPi NEO（Cortex-A7）促销价是 56 元人民币；
- 国内供应商的 i.MX28X（ARM9）促销价是 45 元人民币。

旧款产品往往采用 Linux 最小系统。安装完整版 ARM Linux 不是不可以，但供应商往往不

愿在旧产品上继续支持用户。在 Linux 最小系统中增加一个 MicroPython 是一种很好的升级方式。

6.6.6　MicroPython 库

既然提到了 micropython-pip 和 micropython-lib，那么我们来了解一下 MicroPython 各种库的分类和实现。

6.6.6.1　内部库

内部库用于访问和控制 MicroPython 内部，包括：

- micropython.mem_info([verbose])，打印当前使用了的存储器，包含堆栈和堆；
- micropython.qstr_info([verbose])，打印字符串所使用的信息；
- micropython.alloc_emergency_exception_buf(size)，为紧急 exception 分配 RAM，用于中断处理之类的场景。

6.6.6.2　内置库

内置库是包含在 MicroPython 解释器中的标准库。

- cmath，复数相关库；
- gc，垃圾回收；
- math，算术计算库；
- os，基本文件操作系统库；
- select，在 streams 上等待事件的异步库（目前仅支持 UART/USB_VCP 两种对象）；
- struct，pack/unpack 基础数据类型原语；
- sys，系统相关函数；
- time，事件相关函数。

6.6.6.3　微型库

MicroPython 毕竟是资源受限的平台，许多标准库无法直接使用，所以提供了微型库来作为替代。所有微型库都在标准库前增加 u（micro）前缀。在 micropython-lib 中有不少此类库。

- ubinascii，二进制/ASCII 转换；
- uctypes，访问 C 数据结构；
- uhashlib，哈希算法；
- uheapq，堆队列算法；
- ujson，JSON 编解码；
- ure，正则表达式；

- usocket，套接字模块；
- uzlib，zlib 解压缩。

6.6.6.4 硬件相关库

MicroPython 的官方开发板是 pyboard。相关的库名称为 pyb。这是与特定硬件有关的库。不同硬件的库名称有可能要改。不过基于 STM32F40X 的开发板往往与官方 pyboard 区别不大，通常直接借用 pyb 模块使用。

- pyb.delay(ms)，延时若干毫秒；
- pyb.udelay(us)，延时若干微秒；
- pyb.millis()，返回复位后的毫秒数；
- pyb.micros()，返回复位后的微秒数；
- pyb.elapsed_millis(start)，返回 start 后的毫秒数；
- pyb.elapsed_micros(start)，返回 start 后的微秒数；
- pyb.hard_reset()，MCU 系统硬复位；
- pyb.bootloader()，软件进入 Bootloader 模式；
- pyb.disable_irq()，关闭中断；
- pyb.enable_irq()，启用中断；
- pyb.freq(sysclk, hclk, pclk1, pclk2)，设置或者读取系统时钟配置；
- pyb.wfi()，等待中断；
- pyb.stop()，进入 sleep 模式；
- pyb.standby()，进入 deep sleep 模式；
- pyb.info()，打印硬件版的信息；
- pyb.main(filename)，设置 boot.py 之后运行的主程序名称；
- pyb.mount(device, mount, *, readonly, mkfs)，安装块设备上的文件系统；
- pyb.repl_uart(uart)，设置/读取 REPL 所在的 UART；
- pyb.rng()，返回 30 位随机数；
- pyb.sync()，同步文件系统；
- pyb.unique_id()，返回 MCU 的 12B（96 位）唯一串号。

6.6.6.5 附加硬件相关类

这些类具体如下：

- Accel：加速度计；
- ADC：模数转换器；
- CAN：CAN 总线；

- DAC：数模转换器；
- ExtInt：外部中断事件处理；
- I2C：I2C 总线；
- LCD：带触摸屏的液晶驱动；
- LED：LED 对象；
- Pin：GPIO；
- PinAF：引脚的其他功能；
- RTC：实时时钟；
- Servo：三线制伺服电机驱动；
- SPI：SPI 总线主机驱动；
- Switch：开关；
- Timer：内部时钟；
- TimerChannel：时钟通道；
- UART：双工串行通信；
- USB_VCP：USB 虚拟串口。

6.6.6.6　网络库

物联网时代的网络接口不可少，MicroPython 的网络库内置对于 TI CC3000 以及 Wiznet W5200 的 Wi-Fi 芯片支持，这包含了网络驱动和路由配置。配置后的接口可以通过套接字编程。要使用 network 库的前提是必须构建带网络支持的固件；也就是说，先构建 C/C++源码中的网络代码，然后构建 Python 库。

这里的 CC3000 和 CC3200 是一类芯片。MicroPython 对其支持分为两种角色：一种作为主控，即 pyboard+CC3000；另一种是直接运行在 CC3000/CC3200 内部的 Cortex-M4 中的 self-hosted 模式。也就是说，MicroPython 既可以运行于主控，也可以在 Wi-Fi MODEM 中运行，读者需要区分清楚。

6.6.7　STM32HAL 分支

在 STM32HAL\boards 文件夹下，有如下分支：

- CERB40（STM32F4 FEZ Ceb40）；
- ESPRUNIO_PICO（STM32F401CDU6）；
- HYDRABUS（STM32F401RE 多功能开发工具）；
- NETDRIUNO_PLUS_2；
- PYBLITEV10；

- PYBV10；
- PYBV11（如图 6-15 所示）；
- PYBV3；
- PYBV4；
- NUCLEO_F401RE；
- DISC_F411；
- DISC_F429；
- DISC_F4；
- DISC_F7。

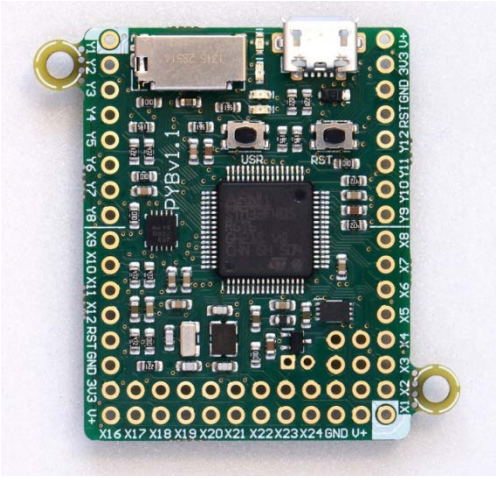


图 6-15　pyboard PYB 的引脚

pyboard 本质上是 STM32HAL 下面的电路板之一。前面提到 UNIX 可以在没有开发板的情况下测试 MicroPython。作为嵌入式 Python，应该选择一种硬件开发板做开发。pyboard 相对比较贵；STM32F401RE 是国内能够买到的开发板，大约为 100 元人民币，而且与 pyboard 的 STM32F405RG 一样都是 QFP64 封装，大部分 I/O 和官方版本差别不大。

虽然 NUCLEO-F401RE 资源与 pyboard 类似，但是其 pyboard 板载 microSD 插槽和 USB Device 接口，则需要读者自己将这些引脚焊接出来。这是它们之间硬件的最大差别了。pyboard 的 USB 是个复合设备，除了 USB CDC/VCP 之外，还有 MSD 的功能。用户可以直接将代码复制到 F401/405 的内部 Flash ROM 或 microSD 文件系统中。如果用户没有自行扩展开发板的 USB 端口，就只能够通过串口上的 REPL 交互环境来评估 MicroPython，而无法将用户代码固化在 ROM 中。

6.6.8　NUCLEO-F401RE 适配

NUCLEO-F401RE 是 STM 推出的廉价开发板，接口与 Arduino 兼容，板载的 STLINK 支持 MSD 下载模式。但 STM 的 NUCLEO 目标板仅仅是一块裸机，缺乏外部晶体和 USB 接口。因此，我们需要做些硬件改动。

6.6.8.1　硬件改动

要激活 NUCLEO-F401RE 板载 USB，需要做的额外工作如下：

（1）焊接 X4/C33/C34 的过孔晶体（8MHz）和表贴电容（22pF）；

（2）将 USB 口从 Arduino 排座和 Morph 插针上引出来转到一个 USB Type-B 插座上。

建议 STM 在以后的 NUCLEO 布板中把 USB 做在上面。因为 USB 几乎是 STM32 的入门级必备功能，所以没有必要省掉。相比之下，STM32F4-Discovery 更加完整，有两个 USB：Debugger USB 和 F407 的用户 USB。可以直接使用目标 MCU 的用户 USB 做评估。

笔者查询了 STM32F401 的数据手册 I/O 布局图，其 USB OTG 端口（DP/DM/ID/VBUS/SOF）与 NUCLEO 开发板的引脚对应关系如表 6-7 所列。

表 6-7　NUCLEO-F401RE 对应的 USB OTG 引脚

名　　称	GPIO	STM32F401RE	NUCLEO-F401RE
DP	PA12	45	CN10-12
DM	PA11	44	CN10-14
ID	PA10	43	D2
VBUS	PA9	42	D8
SOF	PA8	41	D7

实际上，最重要的就是 DP/DM 两个引脚都在 CN10 Morph 插座上。读者若有兴趣的话，可以动手焊接或者做个通用的 NUCLEO 小型转接板。

6.6.8.2　安装工具链

MicroPython 的最初版本基于 STM32F405，其开发基于 UNIX/Linux。必须先下载交叉编译器 arm-none-eabi-gcc。GCC ARM Embedded 实际上是由 ARM 维护的裸机（baremetal）开发专用 GCC 编译器。其具体说明网址如下：https://launchpad.net/gcc-arm-embedded。

```
$ mkdir download
$ cd download
$ wget https://launchpad.net/gcc-arm-embedded/5.0/5-2015-q4-major/+download/gcc-arm
-none-eabi-5_2-2015q4-20151219-linux.tar.bz2
```

Ubuntu 用户可以使用另外一种方式安装：

```
$ sudo apt-get install python-software-properties
$ sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
$ sudo apt-get update
$ sudo apt-get install gcc-arm-embedded （大约 109MB）
```

python-software-properties 的作者还很贴心地将 apt-add-repository 通过符号链接方式指向 add-apt-repository 可执行文件。

如果需要更新 gcc-arm-embedded，请输入：

```
$ sudo apt-get update
$ sudo apt-get install gcc-arm-embedded
```

如果需要移除该组件，请输入：

```
$ sudo apt-get remove gcc-arm-embedded
```

6.6.8.3 从源码构建固件

笔者经常从 GitHub 上直接下载压缩包，解压后进行编译，但压缩包和 git clone 获取的代码好像有时间差。因为 MicroPython 更改得比较频繁，所以建议读者采用 Git 来获取最新源码。

MicroPython 最初使用了 STM32F405RG。STM 在市场上使用 STM32F4XX MCU 的开发板如下：

- STM32F407-Discovery（STM32F407）；
- STM32F411-Discovery（STM32F411）；
- STM32F429-Discovery（STM32F429）；
- NUCLEO-F401RE（STM32F401RE）；
- NUCLEO-F411RE（STM32F411RE）。

以上开发板几乎都可以直接配置运行 MicroPython。MicroPython 固件基于 STM32Cube HAL 库，当前支持大多数 Cortex-M4 微控制器。MicroPython 源码中有关于构建的方法。

```
make BOARD=NUCLEO_F401RE
```

这里的 NUCLEO_F401RE 可以替换成其他所支持的电路板名称，比如 PYBV10、PYBV11 等。如果不指定 BOARD 参数，则会针对 pyboard 编译。不同的 PCBA 的 ROM/RAM、I/O 和 REPL 启动串口是不一样的。接下来我们编译一下 pyboard 的固件。

```
$ cd stmhal
$ make BOARD=NUCLEO_F401RE

......
LINK build-NUCLEO_F401RE/firmware.elf
   text    data     bss     dec     hex filename
 286520     336   27944  314800   4cdb0 build-NUCLEO_F401RE/firmware.elf
```

```
Create build-NUCLEO_F401RE/firmware.dfu
Create build-NUCLEO_F401RE/firmware.hex
```

可以看到用 GCC 生成的 MicroPython 虚拟机，大约占用 ROM 280KB、RAM 28KB。

GCC 默认生成的是 elf 格式文件。要将生成后的固件烧录到 MCU 中去，还需要转换成更加常见的 hex 和 dfu 文件。hex 文件是固件的工业标准格式。ARM mbed 开发板，包括 STM32 NUCLEO 系列开发板往往只接受更原始的 bin 格式。hex 作为常见的固件格式，可以转化为 bin 格式用于下载。DFU 是 USB 组织规定的固件升级类设备。许多手机设备升级固件时会枚举成 DFU 类设备，完成固件升级后再重启，枚举成为其他种类设备（MSD/CDC 等）。

6.6.8.4　固件下载

这里的固件指的是 MicroPython 的解释器固件。我们需要先加载解释器固件，再下载用户 Python 代码。MicroPython 几乎支持了 STM32 的所有主流下载方式。至于 ESP8266 和 TI CC3200 之类的则需要参考文档说明。

1．DFU

我们可以利用 MCU 自带的 USB DFU bootloader，通过 MCU 片内 USB 接口将系统固件烧录到 Flash ROM 中。STM32F4XX 自带 DFU Bootloader，其他一些没有 DFU Bootloader 的 MCU 必须要首先实施该 Bootloader。一般来说，DFU Bootloader 需要占用 32KB ROM 空间。生产后的裸机 PCBA，采用这种方式下载固件很方便。pyboard 就是这样下载固件的。

pyboard 支持 USB DFU 固件设计：

```
$ make deploy
```

或直接使用 dfu-util 来刷固件：

```
$ sudo dfu-util -a 0 -d 0483:df11 -D build-PYBV10/firmware.dfu
```

dfu-util 正是 OpenMoko/OpenEmbedded 工程遗留的宝贵资产，并成为其他工程的常见工具。如果想要一次性构建源码并通过 DFU 下载，可以输入：

```
$ make BOARD=PYB10 USE_PYDFU=0 deploy
```

如果权限不够，请使用 sudo 指令。

2．ST-Flash

STM 提供的各类开发板都有板载调试器 STLINK/V2。板载调试器既可以调试开发板上的 STM32 微控制器，也可以调试其他的客户电路板载 STM32 微控制器。在 Linux/OS X 中，同样可以使用 STLINK/V2 来刷固件，这种方式是通过 MCU 的 SWD 调试口下载固件。首先连接 STLINK/V2 的 USB，输入命令：

```
make BOARD=NUCLEO_F401RE deploy-stlink
```

deply-stlink 会调用 st-flash 程序。此程序依赖于 libusb-1.0-0-dev，来自另外一个开源软件：https://github.com/texane/stlink。

```
$ unzip stlink-master.zip
$ cd stlink-master
$ sudo apt-get install automake
$ sudo apt-get install libusb-1.0-0-dev
$ ./autogen.sh
$ ./configure
$ make
$ sudo ./st-util
$ sudo make install
```

开源软件提供了更多选择。虽然 STLINK/V2 也支持以 U 盘方式的固件下载，但 st-flash 却是从 JTAG/SWD 下载固件。相比前者，JTAG 方式可以很好地控制下载的地址和其他选项，还可以支持 GDB 调试。

Linux 可以自动检测到 STLINK/V2。如果未能够检测到，请使用 lsusb 工具检查，并设置 STLINK_DEVICE 环境变量。

```
$  lsusb
[...]
Bus 002 Device 035: ID 0483:3748 STMicroelectronics ST-LINK/V2
$  export STLINK_DEVICE="002:0035"
$  make BOARD=NUCLEO_F401RE deploy-stlink
```

3．OpenOCD

OpenOCD 是第三方 JTAG/SWD 调试器软件，用于芯片测试、固件调试和烧录，硬件支持 STLINK/V2 调试器。可以通过 OpenOCD+STLINK/V2 组合将 MicroPython 固件烧录到 MCU 中。

注意 OpenOCD 对于 STLINK/V2 固件版本有依赖，所以 STLINK/V2 可能需要升级固件后才支持 OpenOCD。但是 STLINK/V2 只有固件升级工具，没有降级工具。刷了新版固件，可能在旧的开发环境中无法识别，所以读者需要评估更新固件的风险。

```
make BOARD=NUCLEO_F401RE deploy-openocd
```

OpenOCD 的配置文件在 stmhal/boards/openocd_stm32f4.cfg 中。

随后，可以通过 USB 串口或者 UART 连接 MicroPython 运行了。

4．其他方式

许多其他 MCU 还有各自的固件刷新方式。如 mbed MCU 板载 SWD Debugger 最常用的 MSD 方式，可以直接将固件复制到挂载的 MSD 磁盘中。后面提到的 mbed-ls 软件包可以帮助实现这一固件下载功能。还有基于其他类型的 Bootloader 和 OTA 方式，这些完全依赖于硬件供应商 Bootloader/IAP/FOTA 的具体实现。

前面已经介绍了激活 USB 之后的 NUCLEO-F401RE 支持 DFU 下载，本节将采用 ST-Flash 的方式烧录固件。

```
$ sudo make BOARD=NUCLEO_F401RE deploy-stlink

2016-04-02T15:22:16 INFO src/stlink-common.c: Starting verification of write complete
2016-04-02T15:22:19 INFO src/stlink-common.c: Flash written and verified! jolly good!
```

在 NUCLEO-F401RE 中，ST-Link 的 USB VCP 对应于 F401RE 的 UART2，其已经在 boards/NUCLEO_F401RE/mpconfigboard.h 中配置完毕。如果一切顺利，可以在 NUCLEO-F401RE 的 USB 串口中看到 REPL 提示。

```
MicroPython v1.6 on 2016-04-02; NUCLEO-F401RE with STM32F401xE
Type "help()" for more information.
>>> help()
Welcome to MicroPython!

For online help please visit http://micropython.org/help/.

Quick overview of commands for the board:
  pyb.info()    -- print some general information
  pyb.delay(n)  -- wait for n milliseconds
  pyb.millis()  -- get number of milliseconds since hard reset
  pyb.Switch()  -- create a switch object
                   Switch methods: (), callback(f)
  pyb.LED(n)    -- create an LED object for LED n (n=1,2,3,4)
                   LED methods: on(), off(), toggle(), intensity(<n>)
  pyb.Pin(pin)  -- get a pin, eg pyb.Pin('X1')
  pyb.Pin(pin, m, [p]) -- get a pin and configure it for IO mode m, pull mode p
                   Pin methods: init(..), value([v]), high(), low()
  pyb.ExtInt(pin, m, p, callback) -- create an external interrupt object
  pyb.ADC(pin)  -- make an analog object from a pin
                   ADC methods: read(), read_timed(buf, freq)
  pyb.DAC(port) -- make a DAC object
                   DAC methods: triangle(freq), write(n), write_timed(buf, freq)
  pyb.RTC()     -- make an RTC object; methods: datetime([val])
  pyb.rng()     -- get a 30-bit hardware random number
  pyb.Servo(n)  -- create Servo object for servo n (n=1,2,3,4)
                   Servo methods: calibration(..), angle([x, [t]]), speed([x, [t]])
  pyb.Accel()   -- create an Accelerometer object
                   Accelerometer methods: x(), y(), z(), tilt(), filtered_xyz()

Pins are numbered X1-X12, X17-X22, Y1-Y12, or by their MCU name
Pin IO modes are: pyb.Pin.IN, pyb.Pin.OUT_PP, pyb.Pin.OUT_OD
Pin pull modes are: pyb.Pin.PULL_NONE, pyb.Pin.PULL_UP, pyb.Pin.PULL_DOWN
Additional serial bus objects: pyb.I2C(n), pyb.SPI(n), pyb.UART(n)
```

```
Control commands:
  CTRL-A        -- on a blank line, enter raw REPL mode
  CTRL-B        -- on a blank line, enter normal REPL mode
  CTRL-C        -- interrupt a running program
  CTRL-D        -- on a blank line, do a soft reset of the board
  CTRL-E        -- on a blank line, enter paste mode

For further help on a specific object, type help(obj)
>>>
```

笔者在使用过程中，在 Linux 下连接/dev/ttyACM0 时工作不正常，倒是在 Windows 下采用 TeraTerm 可以正常看到 MicroPython 的 REPL。所以大家可以用不同的串口工具进行测试，波特率为 115200bps。ST-Link 的 USB/CDC 串口波特率，使用 9600bps~115200bps 均可。这是因为 USB 虚拟串口的波特率其实没有意义。但在 MicroPython 代码中，与其他外设通信用的硬件串口 UART 波特率必须按照具体情况配置。

6.6.9 pyboard 评估

前面主要为没有 pyboard 的读者提供了几种评估 MicroPython 的方法：

- UNIX 版本 MicroPython 评估；
- 官网提供的 pyboard 评估网页；
- STM32F4XX 的其他开发板评估。

其实在其他的 NUCLEO 上进行 MicroPython 评估非常接近官方 pyboard。但是要评估 MicroPython 的其他特性，如内置文件系统和 TF 卡中的文件系统，必须激活 USB 和 SD Card 接口。

在本章中提供了多种评估方法，主要是由于官方 pyboard 较贵。现在我们评估一下国产的 pyboard。感谢 EEWorld 编辑 nmg 和 dcexpert 提供的国产 pyboard，这让我们得以继续挖掘 MicroPython 的潜力。

EEWorld 版本根据国内 IC 供货情况而优化修改，保留了 George 的大多数原始设计，包括 USB、TF、按键和扩展口，但移除了 3D 加速度计，增加了 LED。如果能够替换成 Invensense 的 6DOF/9DOF 传感器就更完美了，这样可以直接用于可穿戴设计。

6.6.9.1 运行方式

在 pyboard 上，用户大致有三种代码运行的控制方法：

- REPL 交互方式；
- 远程脚本（Raw REPL）；
- 闪存脚本，来自内部文件系统和外部 microSD 卡。

1．REPL 交互方式

将 pyboard 连接到 PC。找到 pyboard 的 USB 虚拟串口（CDC VCP ACM），该串口即 REPL 串口。然后可以通过任何终端程序如 TeraTerm/Putty/minicom 连接该串口，访问 MicroPython REPL。在这种 REPL 命令行模式下，开发者可以直接输入和执行 Python 命令，如同桌面版 REPL 一样。同时，开发者还可以将 REPL 重定向到其他任意一组 UART 中去。但是，首先能够访问的串口是在固件编译时确定下来的。

2．远程脚本

远程脚本即所谓的 Raw REPL 模式。开发者可以在 REPL 中通过 CTRL-A 组合键进入原始 REPL 模式。在这种模式中，开发者可以发送任意 Python 脚本到 pyboard 中，并立即执行。MicroPython 提供一个 Python 脚本简化这一过程。

```
python pyboard.py script_to_run.py
```

该脚本会将 script_to_run.py 发送给 pyboard，并返回执行结果。这种方式应该是为了节省用户逐行输入命令时间而设计的一种模式。本质上是一种通过 USB 虚拟串口传输用户脚本的模式。

3．闪存脚本

pyboard 在 MCU 内部 Flash ROM 维持一个小巧的内置文件系统：/flash。如果内置空间不足，还可以使用外部 TF 卡槽进行扩展。其路径为：/sd。插入 TF 卡后，从文件系统中就看不到 /flash 目录了，而只有外部 TF 卡的内容。

当将 pyboard 插入 PC 后，会显示为一个 U 盘。开发者可以访问内部文件系统和外部 TF 卡中的文件。可以将 Python 脚本改名为 main.py，保存在该文件系统下。这样 pyboard 会在复位后直接执行用户脚本。这也是**唯一**不需要连接 PC 就可以运行 Python 脚本的方式。

4．REPL 和闪存脚本切换

按 CTRL-C 组合键可以中断 REPL 以及 Flash 中运行的程序。这意味着即使用户程序占用 USB 串口的场景，也可以使用 CTRL-C 组合键中断用户程序并切换回 REPL。

当用户代码运行时出现语法错误或出现异常，系统会自动切换到 REPL 模式抛出异常。此时，开发者可以更新 Flash 代码并重新软启动用户脚本。此时，用户脚本可以再次从 REPL 处获得 USB 串口访问权。

5．重启用户脚本

许多工程师，包括笔者本人在使用 pyboard 的时候有个习惯，重启用户脚本时会尝试按下复位按钮。这在 MicroPython 的大多数开发情况下非但没有必要，而且非常危险。硬件复位后发生以下事件：

（1）MCU 重启，重新初始化硬件；

（2）USB 虚拟串口丢失，重新枚举 USB 串口；

（3）REPL 终端因此中断连接，需要重新连接；

（4）重新输入用户代码。

这不仅仅会造成额外的麻烦，最大的风险是可能造成闪存中的文件系统甚至 MicroPython 固件本身丢失！

MicroPython 之所以设计了 CTRL-D/CTRL-C 组合键切换 REPL 和闪存脚本，就是为了避免这一切。

如果出于必要，一定要硬件重启，**则必须在主机中使用安全弹出流程**。虽然有些麻烦，但却可以回避用户代码和固件丢失现象。

6．切换用户脚本

闪存脚本第一个运行的是 boot.py，这是 MicroPython 编译时的默认配置。在 boot.py 中指向用户脚本：main.py，所以不需要反复修改 main.py 内容。可以将自己的新代码保存在另外的文件中，比如 demo.py，并在 boot.py 中指向 demo.py。

如果还想偷懒，直接修改 boot.py 也可以。如何省时省力开发，完全取决于开发者的个人喜好。笔者本人出于评估的需要，会在内部文件夹内放置许多小脚本，用 boot.py 来切换运行。

7．运行模式小结

开发者可以在 REPL、闪存脚本和外置 TF 中运行代码。但是在实际工程中，往往采用内置文件系统保存用户程序，采用在外部 microSD 中记录采集后的数据；且调试过程中可能存在需要切换回 REPL 进行小段代码评估的需求。如果将 USB/UART 从连接到断开视为一次会话，这三种方式可以在一次会话中彼此切换。读者需要厘清不同运行模式间彼此切换的方法：

（1）系统首先启动的是 Bootloader，然后进入 MicroPython 虚拟机。

（2）虚拟机会首先查看文件系统中是否有用户脚本。即 boot.py 是否有内容，如指向用户的 main.py。

（3）如果有，先执行用户脚本。如果没有用户脚本，则切换到 REPL 等待用户输入。

（4）如果用户脚本出错，也会切换到 REPL。

（5）用户可以使用远程脚本发送给 REPL 运行。

（6）用户可以修改文件系统中的脚本，在 REPL 中使用 CTRL-D 组合键软启动重新运行新脚本。

（7）用户可以使用 CTRL-C 组合键中断 REPL 正在运行的代码或正在运行的文件系统中的用户脚本。

6.6.9.2 pyboard 上电使用

pyboard 被插上后，系统可以看到 USB CDC 和 MSD 设备。一般来说，代码固化在 pyboard MCU 内部 flash 中，路径如下：<盘符>/flash。/flash 中有如下四个文件。

- boot.py：pyboard 上电后启动的第一个脚本；

- main.py：包含用户 Python 程序的主脚本；
- README.txt：简单介绍；
- pybcdc.inf：配置 USB CDC 的 Windows 驱动文件。

用户脚本主要在 main.py，打开 main.py，修改代码，保存即可。系统硬件复位后会运行 main.py。这并不意味着所有代码都需要保存在 main.py。读者可以另外定义其他脚本和模块。

1．boot.py

MicroPython 启动后运行的第一个脚本是 boot.py：

```
# boot.py -- run on boot-up
# can run arbitrary Python, but best to keep it minimal

import machine
import pyb
pyb.main('main.py')
```

2．main.py

在 boot.py 中通过 pyb.main(‘main.py’)指定了 main.py 是真正的用户主程序：

```
# main.py -- put your code here!

import pyb

# LED toggle
def LED_toggle():
    for i in range(1, 500):
        pyb.LED(i).on()
        pyb.delay(100)
        pyb.LED(i).off()
        pyb.delay(100)

LED_loop_toggle()
```

以上代码是最基础的 LED 闪烁灯代码。

6.6.9.3　REPL 探索 MicroPython

REPL（Read Evaluate Print Loop）即交互运行模式中的命令读取、计算、打印结果流程。在许多语言中都提供了 REPL 终端：

- Common Lisp；
- Ruby；
- Python；
- Lua；
- Node.js；

- PHP（Facebook phpsh）；
- Java BeanShell；
- C# Mono；
- QBasic。

MicroPython 作为 Python 的分支，同样继承了 REPL。REPL 是通过终端软件连接 USB CDC 串口实现的。

dir 函数

由于 MicroPython 与标准 CPython 存在许多差异，有些甚至在文档里都没有注明，因此需要开发者自己探索。可以在 REPL 中进行简单的测试，最常用的就是 dir。

```
>>> dir(MicroPython v1.6 on 2016-04-02; NUCLEO-F401RE with STM32F401xE
Type "help()" for more information.
>>> dir()
['__name__', 'machine', 'pyb']
>>> dir(__name__)
['encode', 'find', 'rfind', 'index', 'rindex', 'join', 'split', 'splitlines', 'rspli
t', 'startswith', 'endswith', 'strip', 'lstrip', 'rstrip', 'format', 'replace', 'cou
nt', 'partition', 'rpartition', 'lower', 'upper', 'isspace', 'isalpha', 'isdigit', '
isupper', 'islower']
>>> dir(machine)
['__name__', 'info', 'unique_id', 'reset', 'bootloader', 'freq', 'idle', 'sleep', 'd
eepsleep', 'disable_irq', 'enable_irq', 'mem8', 'mem16', 'mem32', 'Pin', 'I2C', 'SPI
']
>>> dir(pyb)
['__name__', 'bootloader', 'hard_reset', 'info', 'unique_id', 'freq', 'repl_info', '
wfi', 'disable_irq', 'enable_irq', 'stop', 'standby', 'main', 'repl_uart', 'usb_mode
', 'hid_mouse', 'hid_keyboard', 'USB_VCP', 'USB_HID', 'have_cdc', 'hid', 'millis', '
elapsed_millis', 'micros', 'elapsed_micros', 'delay', 'udelay', 'sync', 'mount', 'Ti
mer', 'RTC', 'Pin', 'ExtInt', 'Switch', 'Flash', 'LED', 'I2C', 'SPI', 'UART', 'ADC',
 'ADCAll']
>>>
```

6.6.9.4 math 库对比

笔者一直很好奇 MicroPython 的浮点计算。因为桌面版的 Python 浮点一直是双精度浮点数，而 pyboard 使用的 Cortex-M4F 内置 FPU 是单精度浮点数，所以笔者做了一些测试并和桌面版 Python 进行了对比。

Python 3.4 的 math 库：

```
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'at
an2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'ex
```

```
p', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot',
 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', '
pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

MicroPython 的 math 库：

```
>>> import math
>>> dir(math)
['__name__', 'e', 'pi', 'sqrt', 'pow', 'exp', 'expm1', 'log', 'log2', 'log10', 'cosh
', 'sinh', 'tanh', 'acosh', 'asinh', 'atanh', 'cos', 'sin', 'tan', 'acos', 'asin', '
atan', 'atan2', 'ceil', 'copysign', 'fabs', 'floor', 'fmod', 'frexp', 'ldexp', 'modf
', 'isfinite', 'isinf', 'isnan', 'trunc', 'radians', 'degrees', 'erf', 'erfc', 'gamm
a', 'lgamma']
```

MicroPython 的 math 库来自 Python 3.4，它和桌面版 Python 3.4 的最大区别是 MicroPython 仅支持单精度浮点数。Python 与 MicroPython math 库对比表如表 6-8 所示。

表 6-8　Python 与 MicroPython math 库对比表

函　　数	简　　介	Python 3.4	MicroPython 3.4
math.e	自然对数	math.e	math.e
		2.718281828459045	2.718282
math.pi	圆周率 PI	math.pi	math.pi
		3.141592653589793	3.141593
math.sqrt(x)	平方根函数	math.sqrt(2)	math.sqrt(2)
		1.4142135623730951	1.414214
math.pow(x,y)	x 的 y 次方	math.pow(2,10)	math.pow(2,10)
		1024.0	1024.0
math.exp(x)	e 的 x 次方	math.exp(2)	math.exp(2)
		7.38905609893065	7.389056
math.expm1(x)	e 的 x 次方减 1	math.expm1(2)	math.expm1(2)
		6.38905609893065	6.389056
math.log(x)	e 的对数	math.log(10)	math.log(10)
		2.302585092994046	2.302585
math.log2(x)	2 的对数	math.log(2,10)	math.log10(2)
		0.30102999566398114	0.30103
math.log10(x)	10 的对数	math.log(10,2)	math.log2(10)
math.cosh(x)	双曲余弦函数	math.cosh(2)	math.cosh(2)
		3.7621956910836314	3.762196
		3.3219280948873626	3.321928

续表

函　数	简　介	Python 3.4	MicroPython 3.4
math.sinh(x)	双曲正弦函数	math.sinh(2)	math.sinh(2)
		3.6268604078470186	3.62686
math.tanh(x)	双曲正切函数	math.tanh(2)	math.tanh(2)
		0.9640275800758169	0.9640275
math.acosh(x)	反双曲余弦函数	math.acosh(2)	math.acosh(2)
		1.3169578969248166	1.316958
math.asinh(x)	反双曲正弦函数	math.asinh(2)	math.asinh(2)
		1.4436354751788103	1.443635
math.atanh(x)	反双曲正切函数	math.atanh(0)	math.atanh(0)
		0.0	0.0
math.cos(x)	三角余弦函数	math.cos(math.radians(60))	math.cos(math.radians(60))
		0.5000000000000001	0.5
math.sin(x)	三角正弦函数	math.sin(math.radians(30))	math.sin(math.radians(30))
		0.49999999999999994	0.5
math.tan(x)	三角正切函数	math.tan(math.radians(30))	math.tan(math.radians(30))
		0.5773502691896257	0.5773503
math.acos(x)	反三角余弦函数	math.acos(math.sqrt(2)/2)	math.acos(math.sqrt(2)/2)
		0.7853981633974483	0.7853982
math.asin(x)	反三角正弦函数	math.asin(math.sqrt(2)/2)	math.asin(math.sqrt(2)/2)
		0.7853981633974484	0.7853982
math.atan(x)	反三角正切函数	math.atan(0.5773503)	math.atan(0.5773503)
		0.5235987987060793	0.5235988
math.degrees(x)	弧度转度	math.degrees(math.pi)	math.degrees(math.pi)
		180.0	180.0
math.radians(x)	度转弧度	math.radians(45)	math.radians(45)
		0.7853981633974483	0.7853982
math.erf(x)	x 的误差函数	math.erf(1)	math.erf(1)
		0.842700792949715	0.8427008
math.erfc(x)	x 的余误差函数	math.erfc(1)	math.erfc(1)
		0.157299207050285	0.1572992
math.gamma(x)	x 的 gamma 函数	math.gamma(1)	math.gamma(1)
		1.0	1.0

续表

函　数	简　介	Python 3.4	MicroPython 3.4
math.lgamma(x)	x 绝对值的自然对数的 gamma 函数	math.lgamma(1)	math.lgamma(1)
		0.0	0.0

笔者推测 MicroPython 利用了 STM32F4XX 内置 Cortex-M4F 的单精度 FPU，所以 MicroPython 的 math 库也 是单精度的。和桌面系统 Python 采用双精度有所不同，在许多硬件如 CUDA GPU、MCU 中大多采用单精度 FPU。FPU 在一些应用，如飞控 IMU 和可穿戴产品中得到大量使用。单精度大多数情况下已足够了。在其他没有 FPU 的 Cortex-M0/M3 处理器，比如后面提到的 nRF51822 中，MicroPython 的浮点计算应该基于编译器浮点计算库。

笔者在 MicroPython 中测试了简单整数除法，

```
>>>10/3
3.333333
>>>10//3
3
```

测试结果符合 Python 3.X 的定义。而在 Python 2.X 中 10/3 的结果只应该保留整数 3。

数据融合（Data Fusion），指的是对于来自多个传感器的数据进行多级别、多方面、多层次处理，从而提炼出新的有意义信息。这种技术在军事、可穿戴设计、大数据以及物联网的其他应用情景中得到了大量使用。其中，基于传感器的数据融合被称为传感器融合（Sensor Fusion）。

STM32HAL 版本具备了基础的浮点计算能力和足够的计算性能。使用 Python 开发算法，使用复数等数据结构，传感器与融合算法的开发效率会大大提高。

6.6.9.5　文件目录操作

由于 pyboard 的 MCU Flash ROM 中支持文件系统，还有外置 TF 卡槽支持外部 FAT 文件系统，所以 MicroPython 应该支持文件和目录的读写操作。但其官方教程和文档中却一直缺乏相关介绍，只是提到其路径为/flash 和/sd，它们分别对应内置 Flash ROM 文件系统和外置 TF 卡槽。所以笔者猜测它类似于标准 Python 3 的实现。在 Python 2/3 中，和文件、目录操作有关的库主要是 os、os.path 和 shutil。

但 MicroPython 标准库和微型库里面也没有提到最重要的 os 和 sys 模块。我们来做个小小的实验。打开串口终端后，按下 CTRL-D 组合键进入 REPL。

```
PYB: sync filesystems
PYB: soft reboot
MicroPython v1.6-310-g1937953 on 2016-03-28; PYBv1.0 with STM32F405RG
Type "help()" for more information.
```

```
>>>import os, sys
>>> dir(os)
['__name__', 'uname', 'chdir', 'getcwd', 'listdir', 'mkdir', 'remove', 'rename', 'rm
dir', 'stat', 'statvfs', 'unlink', 'sync', 'sep', 'urandom', 'dupterm', 'mount', 'um
ount', 'mkfs']
>>> dir(sys)
['__name__', 'path', 'argv', 'version', 'version_info', 'implementation', 'platform',
 'byteorder', 'maxsize', 'exit', 'stdin', 'stdout', 'stderr', 'modules', 'print_exce
ption']
>>> os.getcwd()
'/sd'
>>> os.listdir('/flash')
['main.py', 'pybcdc.inf', 'README.txt', 'boot.py', 'demo.py', 'modem.py', 'demo2.py',
 'sch_demo.py', 'SPI_demo.py']
>>> os.listdir('/sd')
['HTSC','documents', '.LOST.DIR', 'log', 'bddownload']
>>> fp = open('/flash/boot.py','r')
>>> for line in fp:
...     print(line)
...
...
# boot.py -- run on boot-up
# can run arbitrary Python, but best to keep it minimal
import machine
import pyb
pyb.main('SPI_demo.py')
#pyb.main('main.py') # main script to run after this one
#pyb.usb_mode('CDC+MSC') # act as a serial and a storage device
#pyb.usb_mode('CDC+HID') # act as a serial device and a mouse
>>> fp.close()
>>>
>>> fp = open('/flash/test.txt','w+')
>>> fp.write('hello ')
6
>>> fp.write('world\r\n')
7
>>> fp.write('allankliu')
9
>>> fp.close()
>>> fp = open('/flash/test.txt','r')
>>> for line in fp:
...     print(line)
...
...
...
hello world
allankliu
>>> os.listdir('/flash')
```

```
['main.py', 'pybcdc.inf', 'README.txt', 'boot.py', 'demo.py', 'modem.py', 'demo2.py',
 'sch_demo.py', 'test.txt', 'SPI_demo.py']
>>> os.unlink('/flash/test.txt')
>>> os.listdir('/flash')
['main.py', 'pybcdc.inf', 'README.txt', 'boot.py', 'demo.py', 'modem.py', 'demo2.py',
 'sch_demo.py', 'SPI_demo.py']
```

从 dir 函数来看，MicroPython 实现了 os 库和内置函数 open/close/read/write 等。其可以与桌面 Python 一样，使用 open 函数新建文件，使用 unlink 函数删除文件，使用 rename 函数更改文件名。对于目录的操作主要有 chdir/getcwd/mkdir/rmdir 等函数。所以，在 MicroPython 的/sd 和/flash 文件系统中满足嵌入式系统的基本需求是没有问题的，其使用起来比 C/C++要容易。主要的使用场景如下：

- 可以从文件中读取配置信息（比如 ini/CSV/JSON 格式）；
- 可以向指定文件中保存数据信息以满足数据采集和日志记录需求。

6.6.9.6　USB 通信

大多数 MCU 都只有一个 USB，所以这个 USB 很大概率会与 REPL 复用。可以使用 print() 或者 USB_VCP 模块进行 USB 通信。

```
# main.py -- put your code here!
import pyb
import select

def pass_through(usb, uart):
   usb.setinterrupt(-1)
   led = pyb.LED(1)
   i = 0
   while True:
       usb.write(hex(i))
       usb.write(chr(i))
       usb.write('\t')
       i = (i+1)%256
       led.toggle()
       pyb.delay(100)

pass_through(pyb.USB_VCP(), pyb.UART(1, 9600))
```

MicroPython 还支持 USB_HID 类，这留待读者自己探索吧。

6.6.9.7　SPI 通信

SPI 是许多物联网 RFIC 的默认通信方式。基于 SPI 类，可以构建 CC11XX/nRF24L01/SX1267 等设备子类。笔者在 pyboard 中做过一个 local loop 的测试。将 MOSI 与 MISO（即 X7/X8）短接，如果从 MISO 返回的数值等于 MOSI 的输出，则工作正常。

```
from pyb import SPI
buf = bytearray(10)
spi = SPI(1, SPI.MASTER, baudrate=9600000, polarity=1, phase=0, crc=0x07)
data = spi.send_recv(b'0123456789')
spi.send_recv(b'0123456789', buf)
#spi.send_recv(buf, buf)

print(repr(buf))
```

如果不短接 MOSI/MISO，返回的数值如下：

```
>>>
PYB: sync filesystems
PYB: soft reboot
bytearray(b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff')
```

短接 MOSI/MISO 后返回的数值如下：

```
>>>
PYB: sync filesystems
PYB: soft reboot
bytearray(b'0123456789')
```

返回数值说明，SPI 收发正常。

在 mbed 网站中有许多 RFIC 的 C++类库（CC1101/CC2500/CC2530/CC2650/nRF51/nRF24L01），花费一点时间可以很容易地移植为此类 SPI 设备的 Python 类。

返回数值说明，SPI 收发正常。笔者继续测试了 pyboard 的 SPI 总线速度极限。LoRaWAN 虽然速度不高，但是如果对接 FSK 器件，那么大部分 WSN 的空口速率在 250kbps，而 SPI 速率都在 2Mbps 左右。在此例中，最初的 SPI 速度参数设置为 600000，即 600kbps，距离 SX1276 的 10Mbps 速度还有距离。但笔者将速率设置到 9.6MHz 时，其依然工作正常。

结论：在 MicroPython/Python 中，将 Buffer bytearray 长度设置为 256B，即使工作在较高频率，问题也不大。

6.6.9.8 UART 通信

```
from pyb import UART
uart = UART(1, 9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

同样可以将 RX/TX 短路，返回相同数值即正确。

6.6.9.9 I2C 通信

```
from pyb import I2C
i2c = I2C(1, I2C.MASTER, baudrate=100000)
```

```
i2c.scan() # returns list of slave addresses
i2c.send('hello', 0x42) # send 5 bytes to slave with address 0x42
i2c.recv(5, 0x42) # receive 5 bytes from slave
i2c.mem_read(2, 0x42, 0x10) # read 2 bytes from slave 0x42, slave memory 0x10
i2c.mem_write('xy', 0x42, 0x10) # write 2 bytes to slave 0x42, slave memory 0x10
```

在 EEWorld 的 pyboard 上只有 MCU，没有挂接任何 I2C 总线设备，i2c.scan()返回一个空白列表。在官方版本的 Python 中，会返回 Freescale 的 3D 加速度器的设备地址。读者可以挂接一个最常见的 EEPROM 24CXX 做测试。

6.6.9.10　定时器

```
from pyb import Timer
tim = Timer(1, freq=1000)
tim.counter() # get counter value
tim.freq(25) # 25 Hz
tim.callback(lambda t: pyb.LED(2).toggle())
```

利用定时器以 25Hz 频率闪烁绿灯。这段代码利用 lambda 匿名函数实现了中断驱动的 LED 显示。

6.6.9.11　ADC 数据采集

物联网设备端的主要任务之一是数据采集和传输。ADC 采样是必须测试的。

1．ADC 引脚定义

可查阅 MicroPython 官网文档 MicroPython PVBv1.0 引脚定义中关于 ADC 的引脚定义与位置。

2．ADC 定时采集

查阅文档后，发现 MicroPython ADC 的讲究还是很多的。比如，内置库中的定时器采集是针对某个 ADC 通道的，一个定时器对应一个通道。

```
>>> adc = pyb.ADC(pyb.Pin.board.X19)
>>> tim = pyb.Timer(6, freq=10)
>>> buf = bytearray(100)
>>> adc.read_timed(buf, tim)
100
>>> buf
bytearray(b'>/\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\
x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\
x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\
x03\x03\x03\x03\x03\x03\x03\x04\x03\x04\x03\x03\x03\x03\x03\x04\x03\x03\x03\x03\x03\
x03\x03\x03\x04\x04\x03\x03\x03\x03\x04\x04\x03\x04\x03\x03\x04\x03\x04')
```

这里有几个问题需要提一下：

- adc.read_timed()是堵塞型的。因为采样频率为 10Hz，采样数为 100 组。输入 adc.read_timed()之后，需要等待 10s 才返回 100 字节，所以其仅适合批量采样的场景。
- buf 采用了 bytearray。而 ADC 本身为 12 位，bytearray 为 8 位，这损失了数据精度。所以，不得不采用 Python 中的数组 array。

在此，我们使用 array，并使用长整数来保存 ADC 数据。

```
>>> import array
>>> adc = pyb.ADC(pyb.Pin.board.X19)
>>> tim = pyb.Timer(6, freq=10)
>>> buf = array.array('i',[0 for i in range(100)])
>>> adc.read_timed(buf, tim)
400
>>> buf
array('i', [1001, 166, 100, 105, 104, 100, 99, 98, 102, 103, 98, 98, 101, 102, 98, 101,
101, 99, 100, 96, 93, 98, 99, 93, 92, 98, 94, 97, 94, 90, 96, 96, 92, 90, 94, 92, 90,
89, 88, 93, 93, 90, 87, 92, 92, 93, 87, 91, 86, 85, 88, 90, 85, 84, 87, 85, 82, 88, 82,
86, 87, 81, 81, 85, 86, 80, 84, 84, 84, 84, 80, 78, 83, 83, 80, 83, 82, 78, 82, 78, 76,
80, 80, 76, 75, 78, 75, 80, 74, 80, 81, 77, 74, 79, 78, 74, 78, 77, 73, 78])
```

array 要比 list 节省资源，但笔者的代码利用了列表推导式（list comprehension），还真不好说是否节省了资源。另外一种替代方案是 buf = array.array('i', range(100))，可以使用惰性计算。但是数组的初始值是递归增长的，并非全部为零。不过，这点儿瑕疵可以忽略。

这在少量 ADC 采样时可行，但是多个通道就有点儿耗费定时器了，而且还居然是堵塞式的（除非这个 Timer trigger 可以共享）。与前面 6.5.7.1 节中介绍的 Zerynth ADC 代码相比，MicroPython 的 ADC 采样方法需要进行优化。如果要采集多个模拟数据，最直接的办法就是笔者目前的代码，采用单点采样：

```
adc_x = pyb.ADC(pyb.Pin('X19'))
adc_y = pyb.ADC(pyb.Pin('X20'))
adc_z = pyb.ADC(pyb.Pin('X21'))
adc_a = pyb.ADC(pyb.Pin('Y11'))
adc_b = pyb.ADC(pyb.Pin('Y12'))
adc_c = pyb.ADC(pyb.Pin('X11'))
x = adc_x.read()
y = adc_y.read()
z = adc_z.read()
tilt = adc_a.read()
pan = adc_b.read()
roll = adc_c.read()
```

分别采集加速度计和陀螺仪的三路 ADC，共计六路 ADC。此外的方法是采用 pyb.ADCAll()，然后访问对应的 ADC 通道。

```
>>> resolution = 12 # 12bit ADC
```

```
>>> adc = pyb.ADCAll(resolution)
>>> adc
<ADCAll>
>>> for i in range(16):
...     adc.read_channel(i),
...
...
...
1002 1025 976 1053 512 404 708 901 1075 1058 1304 1125 987 995 1100 1189 941 1485
>>>
```

现在的数据采集示例如下：

```
# main.py -- put your code here!
import pyb
import ujson

led = pyb.LED(1)

adc_x = pyb.ADC(pyb.Pin('X19'))
adc_y = pyb.ADC(pyb.Pin('X20'))
adc_z = pyb.ADC(pyb.Pin('X21'))
adc_a = pyb.ADC(pyb.Pin('Y11'))
adc_b = pyb.ADC(pyb.Pin('Y12'))
adc_c = pyb.ADC(pyb.Pin('X11'))

def setup():
    pass

def loop():
    global led, usb
    global adc_x, adc_x, adc_z
    global adc_a, adc_b, adc_c

    x = adc_x.read()
    y = adc_y.read()
    z = adc_z.read()
    tilt = adc_a.read()
    pan = adc_b.read()
    roll = adc_c.read()

    js = ujson.dumps({'x':x,'y':y,'z':z, 'tilt':tilt, 'pan':pan, 'roll':roll})
    print(js)
    led.toggle()
    pyb.delay(250)

setup()
while True:
```

```
    loop()
```

也可以利用 timer callback 中断服务程序，将其作为多路采集的例子。

```
from pyb import Timer, ADC, Pin
import micropython
micropython.alloc_emergency_exception_buf(100)

x = ADC(Pin('X19'))

def daq(t):
    print(x.read())
    #print(t)

tim = Timer(1, freq=1, callback=daq)
```

以上例程得到过 dcexpert 的技术支持。首先，x = ADC(Pin('X19'))是创建对象，不可以在 ISR 内部，一定要放在外部创建。其次，ISR 函数 daq 一定要放一个参数 t（尽管该参数用不上）。在代码中，笔者也尝试通过 print(t)将这个 t 打印出来。居然是 timer 实例！

```
Timer(1, freq=1, prescaler=3124, period=53759, mode=UP, div=1, deadtime=0)
```

6.6.9.12 JSON 编码

虽然 MicroPython 可以和普通嵌入式设计一样使用二进制通信协议，但由于 MicroPython 支持丰富的数据结构，所以使用标准互联网协议如 JSON 是更加自然的事情。可以直接和服务器端，或通过 USB/UART 之上的 RPC 协议与其他设备交互通信。实际在上面的 ADC 例子中已经可以看到 ujson 的应用了。

```
import ujson
x,y,z = 1,2,3
tilt,pan,roll=100,200,300
js = ujson.dumps({'x':x,'y':y,'z':z, 'tilt':tilt, 'pan':pan, 'roll':roll})
print(js)
```

6.6.9.13 调度器与协程

有读者在使用 MicroPython 的时候尝试使用多线程设计。MicroPython 最初没有支持多线程，之后应开发者的强烈要求在新版本中增加了多线程的支持。嵌入式系统的任务调度和桌面系统中存在较大差异，需要开发者留意。

MicroPython 的作者推荐采用列表来实现一个微型调度器。其实现有些类似于 Python sched 标准库中的调度器。

```
#!/usr/bin/env python
import pyb
```

```
class Scheduler:
    def __init__(self):
        self.events = []

    def schedule(self, delay, function):
        if delay <= 0:
            function()
        else:
            self.events.append([delay, function])
            self.events.sort(key=lambda event:event[0])

    def run(self):
        while len(self.events) > 0:
            delay, function = self.events.pop(0)
            for event in self. events:
                event[0] -= delay
            pyb.delay(delay)
            function()

def job1():
    pyb.LED(1).toggle()
    sched.schedule(1000,job2)

def job2():
    pyb.LED(2).toggle()
    sched.schedule(2000,job1)

sched = Scheduler()
job1()
job2()
sched.run()
```

即使在服务器系统里，异步 I/O 也是主流设计。所以，单线程异步 I/O 模式比较适合嵌入式。由于 MicroPython 实现了 yield，所以可以尝试一下协程的写法。

```
import urandom

def genData():
    data = [i for i in range(4)]
    for i in range(4):
        data[i] = urandom.randint(1,100)
    return data

def consume():
    moving_sum = 0
    data_seen = 0
    while True:
        print("wait for to consume")
```

```
        data = yield
        data_seen += len(data)
        moving_sum += sum(data)
        print('Consumed, average %f'%(moving_sum/data_seen))

def produce(consumer):
    while True:
        data = genData()
        print('Produced %s'%repr(data))
        consumer.send(data)
        yield

consumer = consume()
consumer.send(None)
producer = produce(consumer)

for i in range(10):
    print ('Producing...')
    next(producer)
```

MicroPython 的 yield 语句和桌面 CPython 表现得一样棒！顺便提一下，MicroPython 通过 urandom 支持随机数，开发者可以不用再为随机数生成挠头了。

6.6.9.14 多模块编程

面向对象必然需要实现多个模块。为了测试这个功能，笔者设计了一个 Modem 类，准备日后再升级支持真正的 3G/4G MODEM 的 AT 指令集。将其保存在 flash 根目录的 modem.py 文件中。

```
#!/usr/bin/env python

class Modem(object):
    def __init__(self, ser):
        self.part = "Virtual modem for IoT"
        self.serial = ser

    def setup(self):
        print ("%s:ATE0\r\n"%repr(self.serial))

    def shutdown(self):
        print ("%s:AT+PWRDWN\r\n"%repr(self.serial))
```

然后，重新编写了一个 demo2.py，并将 boot.py 中第一个运行的代码指向 demo2.py。在 demo2.py 中导入了 modem.py 模块。

```
# main.py -- put your code here!
import pyb
```

```
#import select
import ujson
import modem

led = pyb.LED(1)
usb = pyb.USB_VCP()
mdm = modem.Modem('uart1')

def setup():
    pass

def loop():
    global led, usb, mdm
    mdm.setup()
    mdm.shutdown()
    led.toggle()
    pyb.delay(500)

setup()
while True:
    loop()
```

所以，MicroPython 和桌面 Python 在多模块支持上没有区别。其可以开发较为复杂的设计。实际上，micropython-upip 都是以多模块方式提供的。

6.6.10　异步处理和中断处理

因为异步事件处理与中断处理是个难点，所以 MicroPython 的作者对此提供了专门的使用指南。

在嵌入式系统中，采用中断服务例程（ISR）和实时操作系统（RTOS）来实现实时处理。Python 在本质上缺乏实时特性，甚至 CPython 还有一个 GIL 锁导致 CPython 的多线程设计存在重大缺陷。在 MicroPython 中采用回调（callback）和匿名函数（lambda）来实现中断服务。

以 pyboard 中的用户按键为例，具体步骤如下：

```
import pyb
def f():
     pyb.LED(1).toggle()

sw = pyb.Switch()
print(sw())
sw.callback(lambda: pyb.LED(1).toggle()) #1
sw.callback(None) #2
sw.callback(f) #3
```

（1）使用匿名函数开关 LED 显示按键状态；

（2）关闭回调函数；

（3）使用正常函数名实现一样的功能。

从上面的例子中可以看出，MicroPython 允许在 Python 语言中使用回调函数的方式来处理中断。MicroPython 处理中断的最佳实践如下：

- 保持代码尽可能简单、短小。
- 避免在中断服务代码中进行变量存储分配和创建对象，不要添加 list，不要插入 dict，不要使用浮点数。
- 如果 ISR 返回多个字节，可以使用 bytearray。如果需要在 ISR 和主程序间共享整数，可以使用 array。
- 如果 ISR/主程序间有共享数据，可以使用全局变量并设置互斥锁，需要在主程序访问前关闭中断，使用数据后立刻重新打开（即所谓临界区处理）。
- 分配紧急异常处理缓存区以应对 ISR 异常。

这些要求原则上和嵌入式 C 编写 ISR 的最佳实践是一致的，不过 MicroPython 的中断处理有其特殊性。

6.6.10.1 紧急异常处理缓存区

如果 ISR 中出现异常，若没有特别分配的存储器，则 MicroPython 无法产生异常报告。而紧急异常处理缓存区就是为了这一目的而设计的。

```
import micropython
micropython.alloc_emergency_exception_buf(100)
```

6.6.10.2 代码最简化

ISR 代码要尽可能简单、短小，必须做到事件发生后立即处理或设置标志位。实际操作可以延后在主循环中处理。

6.6.10.3 ISR 与主程序间通信

ISR 通常需要与主程序保持通信。整数、字节和字节数组通常可以用于交换数据类型。最简单的方式是通过一个或多个数据对象，通过全局声明或者类进行分享。对此有些特别的限制和风险。

6.6.10.4 对象方法作为回调函数

Python 支持单一 ISR 在实例变量间分享。它同样可以让实现了设备驱动的类用于多台设备实例。比如在两个不同占空比 LED 实例上使用同一个驱动。

```
import pyb, micropython
micropython.alloc_emergency_exception_buf(100)
```

```
class Foo(object):
    def __init__(self, timer, led):
        self.led = led
        timer.callback(self.cb)

    def cb(self, tim):
        self.led.toggle()

red = Foo(pyb.Timer(4, freq=1), pyb.LED(1))
green = Foo(pyb.Timer(2, freq=0.8), pyb.LED(2))
```

这个例子和第 4 章中提到的 C++的 MODEM 设备驱动共享中断服务例程是一回事情。

6.6.10.5　创建 Python 对象

ISR 无法创建 Python 对象实例。这是 MicroPython 的主要限制之一。这是因为 MicroPython 需要从堆中分配存储器。这在中断处理中是不被允许的，因为堆的分配不是可重入的。换而言之，有可能在主程序分配存储器的半路上发生中断。所以为了维持堆的完整性，解释器不许在 ISR 代码中分配存储器。

这个限制的衍生后果是 ISR 无法使用浮点算法，因为浮点类型是 Python 内置对象。类似地，ISR 无法在 list 中添加项目。解决这类问题的途径之一是使用预分配缓存区。例如一个类构建器创建 bytearray 实例和标志位。ISR 方法将数据分配到缓存区中，置起标志位。可以在主程序中分配存储器并实例化。

6.6.10.6　使用 Python 对象

对于类的限制还来源于 Python 的工作原理。Python 代码被编译为字节码执行，每行用户代码通常映射到多个字节码中。当代码被解释器读取后，每个字节码分解为多个本地机器码执行。由于中断可能在机器代码执行的任一点发生，因此极有可能在执行了部分 Python 代码时发生。结果就是如果在主循环中正在修改 Python 对象如 set、list、dict，那么中断发生后就会缺乏内部一致性。

所以，开发者必须非常清楚所修改对象的数据结构。修改内置类型如 dict 可能是有问题的，而修改一个数组或者字节数组没有问题。因为后两者是以单一机器码指令来编写的，中断对其没有影响，即在实时编程中它是原子化的。而用户对象可以实例化整数、数组或者字节数组。

MicroPython 支持任意精度的整数：-2^{30} 到 $2^{30}-1$ 会存储在单一机器字中（32 位），更大的整数必须作为 Python 对象保存。所以大整数操作不是原子化的，不应该在 ISR 中使用。

注意，MicroPython 和 CPython 在 32 位普通整数范围上可能有出入。

6.6.10.7 浮点数的应用

通常我们要避免在 ISR 中使用浮点数。硬件寄存器只接受无符号整数，而将其转换到浮点数常常在主循环中使用。但是，一些 DSP 算法需要浮点数计算。在有硬件浮点计算单元的设备中，例如 pyboard 采用的 Cortex-M4F，我们使用 ARM Thumb assembler（即内联汇编器）来解决这件事情。因为处理器将浮点数保存为机器码，数值在 ISR 和主循环中可以通过浮点数组交换。

6.6.10.8 ISR 异常处理

在 ISR 中如果产生异常，则不会扩展到主循环。除非异常被 ISR 代码处理，中断会被关闭。

6.6.11 中断处理的普遍问题

本节主要介绍实时编程中涉及的话题。程序运行时发生的实时错误特别难以调试，因为这些问题很难再现而且发生概率很随机。所以从一开始就应该按照最佳实践进行设计。

6.6.11.1 中断处理设计

笔者反复提及 ISR 的设计原则是简单、短小，而且必须在一个可预测的短时间内返回控制权。因为当 ISR 运行时，主循环停止运行。由于 ISR 发生的随机性，主循环暂停位置也无法预测。如果 ISR 暂停时间过长或者其返回时长是变化的，则将引起无法诊断的软件 bug。这个问题和中断优先级有一定关联。

如果时间过长，那么低优先级的中断服务会被高优先级抢夺控制权，这意味着在 ISR 中插入了额外的延时。如果高优先级占据过长时间，那么低优先级中断服务无法获得资源。此外还可能出现同一中断源的服务程序，前一个中断服务未结束，而又再次发生中断的情况。

因此，应该避免使用循环结构或尽量最小化。如果没有采用中断处理，则最好避免如磁盘访问、打印语句和 UART 等此类设备 I/O 读写。这些 I/O 读写操作相对缓慢，其持续时间根据内容不同而不同。I/O 操作的另一个问题是文件系统功能不是可重入函数：在 ISR 中使用文件系统 I/O，则主程序会变得非常危险。至关重要的 ISR 代码不应该等待一个事件发生。如果代码可以保证在可预见的时间内返回，这种 I/O 是可以接受的。例如，切换引脚或者 LED。通过 I2C/SPI 访问设备是必要的，但应该计算或测量占用时间，并仔细评估其对应用程序的性能影响。

通常在 ISR 和主循环之间需要的共享数据可以通过全局变量或者通过类或实例变量实现。变量类型通常是整数 int 或布尔 bool 类型，整数数组 array 或字节数组 bytearray，这是因为预分配的整数数组比列表提供更快的访问速度。而且我们必须要考虑哪些数值是可以被 ISR 修改的。尤其是中断发生时，主程序正在访问其中部分数值的情况，这可能导致数据不一致的情况出现。

下面举个例子。ISR 传入的数据存储在一个 bytearray 中，然后将收到的字节数存储到一个整数中，代表准备处理的总字节数。主程序读取这个字节数量，处理字节，然后清理获取的字

节。这时，整个流程可以正常工作。但有可能发生一种情况：主程序刚刚读取字节数量后立即发生中断。ISR 再次把数据添加到缓冲区并更新了数量。但主程序已经读取过了数量，ISR 返回后处理了收到的数据，这种情况下新接收的字节会丢失。

有多种方法可避免这种风险。如果无法使用线程安全的方法，则最简单的方法是使用循环缓冲区。

6.6.11.2　可重入性

如果一个函数或方法在主程序和一个或多个 ISR 之间，或者多个 ISR 之间分享，则可能发生一些危险。这里的问题是，函数本身可能会被新的函数实例所中断。因此，其必须设计成为可重入的函数。在 MicroPython 中尚未提及如何处理，在 C/C++中提到了可重入函数的以下几点特征：

- 不连续调用或持有静态变量；
- 不返回指向静态数据的指针，所有数据由函数调用者提供；
- 通过全局数据本地拷贝来保护全局数据；
- 如果必须访问全局变量，则利用互斥信号来保护；
- 绝不调用任何不可重入函数。

读者遵循以上最佳实践可以避免系统错误。

6.6.11.3　临界区

临界区代码的典型例子是访问受到 ISR 操纵的多个变量的代码。如果中断发生在访问这些变量时，则数值会发生前后不一致的现象。竞争条件是临界区代码的实例：即 ISR 和主程序竞争并尝试改变变量。为了避免不一致性的发生，必须采用某种方式来确保 ISR 期间不改变值的临界部分。方法是在进入该代码区前使用 pyb.disable_irq()关闭中断，最后再使用 pyb.enable_irq()打开中断。

实际上这种方式在 C/C++工程中是非常常见的做法。C/C++往往采用一个宏来关闭和打开中断控制寄存器。首先，某些 CPU 结构进入和退出临界区需要的不仅仅是启停中断控制寄存器，还有其他的系统消耗。其次，CPU 的 NMI 不受中断控制寄存器控制，NMI 事件依然会中断程序执行。最后，在临界区代码执行过程中被屏蔽的中断，在临界区代码退出后仍会再次中断主程序的执行。所以，临界区代码也必须简短。

```
import pyb,micropython,array

micropython.alloc_emergency_exception_buf(100)

class BoundsException(Exception):
    pass
```

```
ARRAYSIZE=const(20)
index=0
data=array.array('i',0 for x in range(ARRAYSIZE))

def callback1(t):
    global data,index

    for x in range(5):
        data[index]=pyb.rng() # simulate input
        index+=1
    if index>=ARRAYSIZE:
        raise BoundsException('Array bounds exceeded')

tim4=pyb.Timer(4,freq=100,callback=callback1)

for loop in range(1000):
    if index>0:
        irq_state=pyb.disable_irq() # Start of critical section

    for x in range(index):
        print(data[x])
        index=0

    pyb.enable_irq(irq_state) # End of critical section
    print('loop {}'.format(loop))
    pyb.delay(1)
    tim4.callback(None)
```

临界区内可以包含单行代码和一个变量：

```
count=0
def cb():# An interrupt callback
    count+=1

def main():# Code to set up the interrupt callback omitted
    while True:
        count+=1
```

上面的例子演示了一个微妙的bug来源。主循环中的 count += 1 带有特定的竞争条件风险，被称为读-修改-写。这是实时系统中的经典 bug。在主循环中，读取 t.counter 的数值，计数器加 1，然后再写回去。在极少数情况下，中断发生在读取之后、写回之前。在中断过程中，t.counter 数值被修改，但是该数值被主循环代码覆盖。此类错误的发生概率很低，很难捕捉，但是不可以忽视。

综上所述，应该特别注意那些在主循环代码中修改 Python 内置类型，而在 ISR 访问这些实例的代码。修改变量的代码应该被视为临界区代码，以确保 ISR 运行时实例处于有效状态。

对于不同 ISR 间共享数据集需要特别注意，其风险在于在低优先级部分更新共享数据时有可能发生高优先级的中断。处理这种情况可以使用互斥对象进行控制。

在临界区中禁用中断是非常常见的，而且是最简单的方式。但是它不仅仅是关闭了容易造成问题的中断，而是将所有的中断都关闭了。通常不倾向禁用中断太久。在中断持续期间，如果发生回调，它就会引入额外的时间变化。在设备中断的情况下，可能导致中断服务反应太迟，继而造成数据丢失或者过载等错误发生。和 ISR 类似，临界区代码的执行时间必须可预测且足够短。

互斥可以用于处理临界区，这可以大大减少禁用中断的时间。主程序在运行临界区之前锁定互斥对象，并在执行后打开互斥锁。ISR 测试互斥对象是否锁定。如果锁定，它避免执行临界代码并返回。设计的挑战在于如何定义 ISR 在无法访问临界变量时的应对策略。

互斥对象的一个简单例子可以在 micropython-sample 工程中找到。网址：https://github. com/peterhinch/micropython-samples.git。

在该代码中，互斥代码禁用了中断，且只需八个机器指令。这种方法的好处是，其他中断受影响的概率降到最低。

6.6.11.4　bytearray 的使用

Python 的列表类型是非常强大但却非常耗费资源的类型。而在嵌入式中大多数采用字节类型的寄存器，所以推荐使用 bytearray 来实现数据缓存等。bytearray 使用前需要事先声明，所占用的 RAM 空间是可以预测的。对于超越 8 位字节宽度的情况，如 12 位/24 位-ADC，有必要采用 array.array()构建 array 类型来替换 bytearray。

6.6.12　使用心得

MicroPython 的确针对嵌入式做了许多优化，如整数的优化、内联汇编、中断服务等；同时还保留了 Python 作为高级抽象语言的一些特性，比如 yield、generator 和装饰器等。扩展库也很吸引人。不过在投入实际工程设计中，需要根据硬件规划 ISR 服务和主体程序结构，包括主程序和回调函数之间的配合，以及主程序中不同模块和任务之间采用 yield 进行协程间的切换。

经过对于 pyboard 的摸索，读者可以掌握源码构建、下载、运行、测试，并构建完整应用的过程。在 6.6.9 节中提供的 ADC 定时采样、文件访问、USB 通信代码已经可以让读者构建一个完整的数据记录仪的应用，涉及数据采集、数据存储和数据传输。读者可以以此为基础开发更多应用，并可以参考现有一些开源物联网应用如 WiPy/LoPy 中 MicroPython 的程序和软件结构来增加 Wi-Fi/BLE 通信、低功耗管理等功能，这样可以少走很多弯路。

前面提到的高阶函数用于批量处理数据的方法，部分内置函数如 map/reduce/filter 在 MicroPython 中还缺乏实现。其他的进阶用法最好在 REPL 里预先测试评估后再投入实际使用。

6.6.13 商品化与知识产权

不少读者会提到 MicroPython 的商品化和软件的知识产权保护问题。MicroPython 的作者目前开始为 ESA（Euraopean Space Agency，欧洲空间局）开发一个 MicroPython 项目用于航天项目。该项目基于 SPARK CPU 架构。笔者猜测其可能基于 UNIX/Linux 平台，而非 MCU 平台。所以，MicroPython 的实用化程度是得到许多专业机构承认的。

MicroPython 大部分情况下以源码形式运行，这的确无法保护源码。笔者认为只要把自己的核心知识产权以 C API 或者套接字服务的形式封装起来就可以实现源码保护了。还有一种方式，就是将内部闪存修改成只可以写入而无法读取的文件系统，这一点在 FAT 文件系统读取函数中修改一下即可。不过这两者都需要底层的修改。在本书编写之际，MicroPython 支持将 Python 源码编译成字节码运行的方式，也可以部分解决此类问题。所以，读者不必过于纠结。

6.6.14 BBC microbit

英国在科学教育方面一直走在前列，最典型的例子就是 Raspberry Pi 的成功推广普及。随后，BBC 也对外公布了 micro:bit（商品名中有冒号，以下简称为 microbit）控制板。其计划是针对 100 万名英国儿童（7 年级或 11—12 岁）免费提供硬件编程的学习机会。

根据 BOM 估算，microbit 的硬件成本至少在 10~15 美元左右。100 万名儿童意味着硬件投入在 1000 万美元。如果再计入加工和物流成本，总成本就更高了。最初的几批 microbit 的确是以免费方法发放的。现在，其通过 element14 等目录分销商进行销售。希望我国的企业也有这种魄力和能力培养下一代，反哺社会。

microbit 开发板如图 6-16 所示，其外形为 4cm×5cm，采用 ARM Cortex-M0 处理器、板载加速度计和磁场计，支持蓝牙和 USB 连接，为 25 组 LED 和两组按键。其合作伙伴阵容强大。

- 微软：为该设备定制了 TouchDevelop 平台，并负责托管用户代码和负责教师培训材料；
- 兰卡斯特大学：开发设备运行时（runtime）；
- 易络盟：负责设备制造；
- Nordic 半导体：提供 nRF51822 BLE 控制器作为主控处理器；
- NXP/Freescale 半导体：提供 KL26Z USB MCU 作为 Debugger，并提供 MMA8652/MAG3110 传感器；
- ARM 控股：提供 mbed 开发和编译器；
- Technology Will Save Us：设计物理外观（由于缺乏设计细节，因此笔者还没有充分体会该 PCB 的某些特殊布局用意）；
- 巴卡莱：负责产品交付；
- 三星：开发 Android APP，并连接到手机和平板；

- Wellcome 基金：提供教师和学校的各种培训机会；
- ScienceScope：开发 iOS APP，并将设备交付给学校；
- Python 软件基金会：将 MicroPython 引入设备。

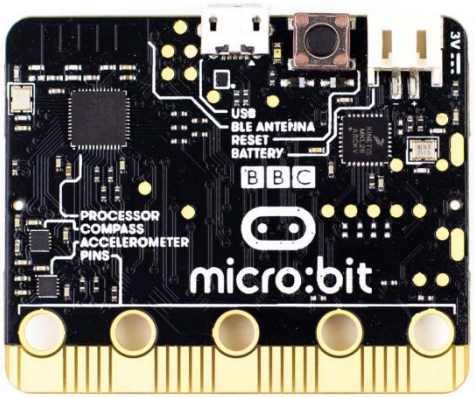


图 6-16　BBC microbit 开发板

6.6.14.1　软件配置

microbit 的软件配置丰富多彩，包括 JavaScript、Blockly、Python 和 C++。其官网信息显示 microbit 支持 4 种编辑器：CodeKingdoms 用于 JavaScript 编程；微软 Block editor 用于 Blockly 编程；微软 TouchDevelop 则可以让客户在手机、平板和桌面电脑上开发编程；MicroPython 用于嵌入式编程。此外，越来越多的第三方编辑器也支持 microbit 编程。

microbit 使用 ARM mbed C++ API 开发基础固件，可以通过 USB（KL26Z SWD debugger）或者蓝牙（nRF51822 BLE FOTA）进行固件刷新。

6.6.14.2　硬件架构

microbit 板载两枚 Cortex-M0 处理器，一枚为 Freescale 的 KL26Z，另外一枚是 nRF51822 主控制器。nRF51822 内置 256KB Flash ROM，16KB RAM，除去 BLE 堆栈消耗，用户空间为 128KB/10KB。microbit 整体架构参考了 ARM mbed 开发板的架构：SWD Debugger + 目标 MCU。

KL26Z 内置 16KB RAM，上电枚举为 mbed SWD 调试器、MSD 网盘编程器以及 CDC 虚拟串口复合设备。程序编译后 hex/bin 文件可以直接拖曳复制到 nRF51822 中去。这意味着 Python 不是运行在 KL26Z 中，而是运行在 nRF51822 内置的 Cortex-M0 中，且和 BLE stack 共享 RAM/ROM 资源。

microbit 上的 Python 也是 MicroPython 分支，最初的版本由 MicroPython 的作者 Damien 负责移植。但是其导入的类不再是 pyb，而是 microbit。也由于 microbit 的硬件采用了 Cortex-M0，

性能明显弱于 pyboard 所采用的 Cortex-M4F，同时 nRF51822 是与 BLE 堆栈共享空间的，所以其在许多方面受到了更多的限制。Nordic nRF52 系列 BLE5 芯片基于 Cortex-M4F，可用资源和处理能力大大增强，期待 microbit 可以移植到 Nordic 的新一代 BLE 芯片中。

microbit 证明：MicroPython 可以运行于入门级 MCU，最小需求是 128KB ROM 和 10KB RAM。大多数 Cortex-M0/M0+均可以满足此项要求。事实上大多数 BLE 芯片的存储器配置是类似的，只不过运行 MicroPython 需要更多计算资源。

百度创新实验室提供的 DuBand 开源手环同样使用 nRF51822，或许可以基于开源硬件和开源 MicroPython 构建一个运行 Python 应用的手环（见图 6-17）。需要通过 OTA 更新固件以支持 MicroPython。由于 MicroPython 需要足够的 RAM，所以推荐使用 nRF51822-QFAC 版本。

图 6-17　笔者在市场上所购的百度手环

部分百度手环出于成本考虑，使用 TI CC2540 BLE SoC 作为主控。该芯片内核为 8051 内核，其无法支持 MicroPython。

6.7　Linux 与 Python

在资源受限的 MCU/MPU 中使用 Linux 的最初目的是：充分利用现成的 GUI、文件系统、通信堆栈和其他系统服务。

传统意义上的嵌入式 Linux 软件大多数都是 C/C++编程，包括机顶盒、电视机等的 UI 界面和控制逻辑也大多采用 C/C++或 Java 编写。

随着越来越多的动态语言和脚本语言的流行，开始出现利用这些语言构建的 UI 和服务。这样，操作系统和核心服务一旦构建完毕，就可以成为一个稳定的平台，而高层 UI 和业务逻辑可以交由客户使用这些语言进行开发，实现产品差异化。这为产品多样化和平台化提供了很好的途径。

6.7.1　Linux 中 Python 的运行环境

Linux 从一开始是针对桌面 x86 处理器的，由于 GCC 的交叉编译支持，因此 Linux 增加了

对于其他处理器架构的支持，并通过剪裁内核逐渐增加对资源有限的嵌入式 MCU/MPU 的支持。

Python 在 Linux 中与 Python VM 的实现有关。Python 被多种语言所实现，所以 Python 在 Linux 中的开发手段是多样的，具体方式如表 6-9 所示。

表 6-9　Python 在嵌入式 Linux 中的开发手段一览表

操作环境	语　　言	虚拟机	虚拟机构建	PyPI
Linux 最小系统	GCC 交叉编译	CPython	交叉编译	PyPI
Linux 最小系统	GCC 交叉编译	PyMite	交叉编译	N/A
Linux 最小系统	GCC 交叉编译	MicroPython	交叉编译	upip
Linux 最小系统	Jython	Java	下载安装	Java+PyPI
嵌入式 Linux 发行版	GCC 交叉编译	CPython/Java	下载安装	PyPI
全功能 Linux 发行版	GCC 原生编译	CPython/Java	下载安装	PyPI

6.7.1.1　LFS/CLFS

前面已经介绍过 LFS/CLFS 是最基础的 Linux 构建方法，其最初的目的是为了在最节省资源的情况下使用 Linux 服务。所以 LFS/CLFS 构建的 Linux 与全功能 Linux 发行版不同，其缺少 Python 运行环境的基本依赖条件。

在各种 Python 运行环境的构建方法中，从源码交叉编译标准版 CPython 是最基础，也是最困难的方式。开发者可以使用替代方法来简化开发，例如交叉编译 PyMite/MicroPython，可以安装在小型 Linux 系统中。如果 Linux 已经安装 Java 运行环境，可以使用 Jython 运行 Python 模块和脚本。

6.7.1.2　嵌入式 Linux 发行版

开发者自行交叉编译 CPython，费时费力、容易出错。比较常见的嵌入式 Linux 发行版，包括 OpenMoko/OpenEmbedded、OpenWRT、OpenZaurus、Angstrom、Familiar、SlugOS 等，可以通过软件管理包来安装 CPython 和 Java 运行环境。

以 OpenWRT 为例，作为面向路由器和网络设备的嵌入式 Linux 发行版，其本质上依然是 Linux 最小系统，是满足特定需求而定制的嵌入式版本，不属于完整 Linux 发行版。由于这些发行版也支持 opkg，所以 OpenWRT 已经提供了交叉编译好的 Python，可以相对简单地安装 Python。在 OpenWRT 中安装 Python 的流程如下：

```
# wget -c http://downloads.openwrt.org.cn/backfire/10.03.1/brcm63xx/packages/libffi
_3.0.9-1_brcm63xx.ipk
# wget -c http://downloads.openwrt.org.cn/backfire/10.03.1/brcm63xx/packages/python
-mini_2.6.4-3_brcm63xx.ipk
# wget -c http://downloads.openwrt.org.cn/backfire/10.03.1/brcm63xx/packages/python
_2.6.4-3_brcm63xx.ipk
```

```
# opkg install libffi*.ipk
# opkg install python-mini*.ipk
# opkg install python_2*.ipk

# wget http://pypi.python.org/packages/2.6/s/setuptools/setuptools-0.6c11-py2.6.egg
# sh setuptools-0.6c11-py2.6.egg
# easy_install pip
```

虽然不像 Ubuntu 中使用 apt-get 那么简单，但已经减少了不少工作量。在 OpenEmbedded 工程中，有关于如何配置 Python 菜单，以及常见 Python 扩展库的说明。读者需要按照某个平台，比如针对 Beagle 的 Angstrom 配置一次，充分体验一下 OpenEmbedded 的配置流程。

6.7.1.3 全功能 Linux 发行版

随着时间和市场的变化，Linux 的许多要素也在持续变化：

- 开发者需求越发复杂——要求支持各类脚本编程以及各种新协议等；
- 某些系统组件使用交叉编译方式非常复杂而容易出错；
- 硬件平台成本的快速下降——手机和平板电脑的配置越来越高，海量存储器和云服务成本持续降低；
- 硬件平台功耗快速下降；
- Debian/Ubuntu/Fedora 发布针对 ARM/MIPS 等非 x86 架构的版本。

在 Debian/Fedora 基础软件库上，针对不同架构的全功能 Linux 发行版日渐流行，包括：

- Fedora for ARM；
- Debian for ARM；
- Ubuntu for ARM；
- Ubuntu Core；
- Ubuntu Mate；
- Pidora/Raspbian。

支持这些全功能版 Linux 的廉价 Linux SBC 越来越多，成本越来越低，系统迭代开发周期越来越短。使用这些平台上的原生 GCC 就可以编译本平台的代码，而无须宿主机参与，可以避免交叉编译的麻烦。在这些平台上，可以直接安装运行 CPython 虚拟机和 Java VM 环境，与桌面系统一致。

然后这一切的基础依然是 LFS/CLFS。接下来我们讲解一下在最小系统中构建 Python 运行环境的步骤。

6.7.2　交叉编译 CPython

使用交叉编译器从源码构建 Python 运行环境是可行的。但是 Python 并没有针对交叉编译做过优化，所以较为烦琐。Python 解释器的构建方式依赖于：

- Python 版本和位数；
- Python 标准库及其依赖的 SQLite 等软件的剪裁；
- 交叉编译器以及交叉编译器的 libc 版本。

交叉编译往往依赖于 diff 补丁包，因此，一旦依赖项目版本升级，则需要开发者重新摸索，寻找或自行制作 diff 补丁包。本节内容仅供参考，在特定的平台上可能会有编译和运行错误，需要读者自行解决。

Python 交叉编译流程如下：

（1）下载源码和软件补丁；

（2）裁剪所需的 Python 模块；

（3）编译出 Python 和 Parser/pgen，后两者会在交叉编译阶段使用。

（4）使用交叉编译器二次编译 Python、Parser/gpen 和其他库。

接下来我们以 PowerPC 平台上的 Python 2.7.5 移植为例介绍具体操作。虽然嵌入式系统的特点是多样性，但其实移植的平台基本上都是 ARM9/MIPS 等小资源 Linux 板，具备通用性。其实可以做一些补丁和脚本进行自动化处理。

6.7.2.1　Python 2.7.5 和补丁

首先需要安装合适的交叉工具链（gcc）、Python 2.7.5 源码和对应的交叉编译补丁。

```
$ export RFS=/path_to_embedded_root_file_system
$ wget http://www.python.org/ftp/python/2.7.5/Python-2.7.5.tar.bz2
$ tar -jxf Python-2.7.5.tar.bz2
$ cp <patch_folder>/Python-2.7.5-xcompile.patch Python-2.7.5/
$ cd Python-2.7.5
```

6.7.2.2　编译构建所需程序

接下来我们要编译出 Python 解释器和 Parser/pgen。所谓 Parser/pgen 是 Parser 目录中的 pgen 语法分析器，用于处理语法定义。

```
$ ./configure  # for build-system's native tool-chain
$ make python Parser/pgen
$ mv python python_for_build
$ mv Parser/pgen Parser/pgen_for_build
```

将编译产生的 Python 解释器和 Parser/pgen 保存起来，为后来的交叉编译和安装做准备。

6.7.2.3 准备交叉编译

保留前面产生的 Python 解释器和 pgen 文件，其余文件删除。然后使用交叉编译器重新配置、编译，产生全套的 Python 库，包括 ssl、zlib、ctypes。

```
$ export PATH="/opt/freescale/usr/local/gcc-4.3.74-eglibc-2.8.74-dp-2/powerpc-none-
linux-gnuspe/bin:${PATH}"
$ make distclean
$ ./configure --host=powerpc-none-linux-gnuspe --build=i586-linux-gnu --prefix=/ \
    --disable-ipv6 ac_cv_file__dev_ptmx=no ac_cv_file__dev_ptc=no  \
    ac_cv_have_long_long_format=yes
```

以上是针对 Freescale PowerPC 的交叉编译器配置。针对 ARM/MIPS 等平台需要配合合适的交叉编译器。

6.7.2.4 交叉编译

重新配置工具链后进行交叉编译：

```
$ make --jobs=8 \
    CFLAGS="-g0 -Os -s -I${RFS}/usr/include -fdata-sections -ffunction-sections" \
    LDFLAGS='-L${RFS}/usr/lib -L${RFS}/lib'
```

最后为编译后的 Python 可执行文件“减肥”。嵌入式系统资源有限，文件尺寸小点儿是必要的：

```
powerpc-none-linux-gnuspe-strip --strip-unneeded python
```

安装 Python：

```
$ sudo make install DESTDIR=${RFS} PATH="${PATH}"
```

针对 Freescale 平台交叉编译 CPython 的内容引自 *Trevor's IT Blog*。英文网址可参见本章延伸阅读部分。此外，视目标系统情况，还需要预先交叉编译 SQLite。同时，并不是所有 Python 版本都有对应的交叉编译补丁包。衷心希望 Python 在以后的版本中能够针对交叉编译进行优化，这样可以推动 Python 在嵌入式 Linux 中的普及使用。

6.7.3 交叉编译 MicroPython

笔者曾经认为，MicroPython 本来就是在 x86 上开发 ARM Cortex 架构的 Python，天生就支持交叉编译，所以 MicroPython 运行在 Linux 板上是件顺理成章的事情。事实上是，虽然 UNIX makefile 里面预留了交叉编译的选项，但实际上没有指定交叉编译器和相关配置。所以，需要读者自己探索一下。

在嵌入式版本的 MicroPython 路径中，打开 minimal/Makefile。这里明确地定义了交叉编

译器：

```
ifeq ($(CROSS), 1)
CROSS_COMPILE = arm-none-eabi-
endif
```

再看其 GitHub 源码中的 unix/Makefile：

```
# On OSX, 'gcc' is a symlink to clang unless a real gcc is installed.
# The unix port of micropython on OSX must be compiled with clang,
# while cross-compile ports require gcc, so we test here for OSX and
# if necessary override the value of 'CC' set in py/mkenv.mk
ifeq ($(UNAME_S),Darwin)
CC = clang
# Use clang syntax for map file
LDFLAGS_ARCH = -Wl,-map,$@.map -Wl,-dead_strip
else
# Use gcc syntax for map file
LDFLAGS_ARCH = -Wl,-Map=$@.map,--cref -Wl,--gc-sections
endif
LDFLAGS = $(LDFLAGS_MOD) $(LDFLAGS_ARCH) -lm $(LDFLAGS_EXTRA)
```

接着查看 py/mkenv.mk：

```
AS = $(CROSS_COMPILE)as
CC = $(CROSS_COMPILE)gcc
CXX = $(CROSS_COMPILE)g++
LD = $(CROSS_COMPILE)ld
OBJCOPY = $(CROSS_COMPILE)objcopy
SIZE = $(CROSS_COMPILE)size
STRIP = $(CROSS_COMPILE)strip
AR = $(CROSS_COMPILE)ar
ifeq ($(MICROPY_FORCE_32BIT),1)
CC += -m32
CXX += -m32
LD += -m32
```

将 unix/Makefile 中的 CC 配置成读者自己的交叉编译器即可：

```
ifeq ($(CROSS), 1)
CROSS_COMPILE = arm-linux-
endif
```

然后进行交叉编译：

```
$ cd unix
$ make
```

如果编译有问题，请提交给 GitHub 以获得进一步支持。笔者接下来的计划之一就是针对国

内的某些廉价开发板评估 CPython 2.7.X/3.X，以及 MicroPython 的运行情况。

6.7.4 Jython 运行环境

Jython 的运行环境本质上就是 JRE（Java Runtime Environment），即运行 Java 程序所必需的环境的集合，包含 JVM 标准实现及 Java 核心类库。不过，如何移植 JRE 不属于本书范畴，请参考 Oracle 的相关文章了解详情。增加此节的主要原因还是要打破大家的固定思维方式，让大家了解 Python 是如何在嵌入式的 Java VM 中运行的。Jython 本身的移植与 CPython 一样，也有交叉编译问题。

Oracle/Sun Java 针对嵌入式系统有许多版本，其中 Java SE Embedded 和 Java SE Embedded Suite 比较适合高性能 Linux SBC。后者包括了 Java SE Embedded、Glassfish、Web Service 等。

我们还是以大家最熟悉的树莓派为例，Raspbian 中已经内置 JRE，所以不必再去 Oracle 网站下载了。先查看以下 Java 的版本：

```
pi@raspberrypi $ javac -version
javac 1.7.0_40
```

6.7.4.1 Pi4J

Pi4J 提供了面向对象风格的 I/O 访问 API 和 Java 类库，可以充分挖掘树莓派的 I/O 能力。该工程将原生 I/O 和中断服务抽象化，Java 程序员可以专注于高层应用业务逻辑。Pi4J 支持树莓派、香蕉派和类似的 Linux SBC。Pi4J 还提供了一些演示程序，比如 GPIO、串口、系统和网络等。

安装

在树莓派控制台输入以下命令：

```
wget http://pi4j.googlecode.com/files/pi4j-0.0.5.deb
sudo dpkg -i pi4j-0.0.5.deb
/opt/pi4j/lib
/opt/pi4j/examples
cd code/java
javac -classpath .:classes:/opt/pi4j/lib/'*' -d . Test.java
sudo java -classpath .:classes:/opt/pi4j/lib/'*' Test
```

Java 代码太长了，请移步本章延伸阅读部分自行下载代码。

6.7.4.2 TigerJython

TigerJython 是 Jython 的一个分支。TigerJython 的官网为 http://www.tigerjython.ch/。其开发团队位于瑞士，他们写了一本教程：*Programming Concepts in Python with TigerJython*。

虽然面向编程新手，但 TigerJython 的读者群的技术栈要求挺高：

- 熟悉嵌入式 Linux，尤其是树莓派；
- 熟悉 Java，因为 Jython 要访问 Java 类库；
- 熟悉 Python，因为编程语言是 Python；
- 了解 Jython。

通常来说，Python/Java 程序员会使用树莓派中的原生环境进行编程，通过 Jython 来操作硬件的做法相当小众。笔者之所以介绍嵌入式 Jython 的主要原因是某些 Linux SBC，如 Freescale 的某些产品是 Linux + Java 配置，使用 Jython 可以直接运行 Python 脚本。

TigerJython 有单独的下载网址：http://jython.tobiaskohn.ch/。

6.7.4.3　Jython 示例

从官网例子来看，TigerJython 的应用主要集中在绘图、游戏制作。

tigerjython_demo.py：

```
from gturtle import *
makeTurtle()

setPenColor("red")
right(45)
repeat 4:
    forward(120)
    left(90)
```

以上例子，读者可能觉得不太像 Python 脚本。在 TigerJython 的文档中，说明 TigerJython 与 Python 存在一些差异：

- 增加 repeat 关键字做迭代控制；
- 修改了 input/print 函数的定义，从原来的终端变成面向对话框的函数；
- 添加了针对多媒体的内置函数。

TigerJython 操纵硬件 GPIO 等，需要导入 RPi_GPIO 类库。该类库位于 TigerJython 的类库中。

tigerjython_gpio.py：

```
from RPi_GPIO import GPIO

# pin numbers
switch = 26
led = 12

print "Press button turn blinking on/off"

GPIO.setup(led, GPIO.OUT)
GPIO.setup(switch, GPIO.IN, GPIO.PUD_UP)
buttonPressed = False
```

```
blinking = False
ledOn = False

while True:
    v = GPIO.input(switch)
    if not buttonPressed and v == GPIO.LOW:
        buttonPressed = True
        blinking = not blinking

    if buttonPressed and v == GPIO.HIGH:
        buttonPressed = False

    if blinking:
        if ledOn:
            ledOn = False
            GPIO.output(led, GPIO.LOW)
        else:
            ledOn = True
            GPIO.output(led, GPIO.HIGH)
    else:
        ledOn = False
        GPIO.output(led, GPIO.LOW)
    GPIO.delay(100)
```

这段代码看着就与标准 Python 一样了。

6.7.5 Android SL4A

谷歌 Android 设备的出货量比 iOS 设备多。谷歌针对 Android 推出 SL4A 工程，即 Script Language for Android。该项目与 Android Scripting Environment（ASE）意义相同。其目的是为 Android 平台提供各类脚本语言（包括 Python、Perl、JRuby、Lua、BeanShell、JavaScript、Tcl、shell 等脚本语言）的编程能力。

在实际开发中，我们不需要同时使用这么多的脚本语言。可以下载单独的 Python 版本 QPython。该 APP 包含了 SL4A、控制台、编辑器、库管理器，支持 Kivy、Web 和 Console 的三大类演示程序。为了解决开发者在虚拟键盘上输入和代码分享的困难，还支持 FTP、网页编辑器和 QR 代码传输 Python 代码。SL4A/QPython 的主要应用场景如下：

- SL4A Android 应用编程，可以利用 Python 进行脚本编程。
- 利用 Django 进行网络编程和小型 Web 开发。
- 使用 Python 脚本和 Kivy 编程环境进行游戏开发。

QPython 让 Android 可以方便地运行 Python 程序（不仅仅是玩具或学习工具）。它还可以在手机上实施 Web。配合 Python 的图像、运算、网络和第三方库支持，Python/Web 开发者可以在

很短的时间内很容易地开发体验良好的 Web。其开发和迭代速度要快于原生 Java 代码。

打开 QPython 的库管理器，可以看到和手机相关的库，有比较通用的 Kivy、PIL、pygame。此外，还可以通过 PyPI 安装纯 Python 编写的各类第三方库。因为 Android 平台上没有 C 编译器，所以必须是纯 Python 模块。

笔者一直猜测 QPython 究竟使用 CPython 还是 Jython 方式，启动终端后发现：

```
Python 2.7.2 (default, Nov 2 2015, 01:07:37)
GCC 4.9 20140827 (prerelease) on linux4
Type "help", "copyright", "credits" or "license" for more information
>>>
```

笔者觉得这个终端信息误导了大家。因为 SL4A 脚本引擎实际上是 Java 程序，而非使用 GCC 构建的 CPython。

SL4A 的 Python 解释器源码都是开源的，由于其采用了 API facade，所以不同 Android 版本的权限设计变化，需要开发者经常去重新构建不同版本的解释器。

关于 Android 平台上 SL4A/QPython/Django/Kivy 的更多细节，笔者会在第 7 章再做更进一步的介绍。

6.8　本章小结

本章重点介绍了 Python 在各类嵌入式设备中运行环境的实现、剪裁、交叉编译和安装使用。PyMite/pymbed/MicroPython/Zerynth 都可以将 Python 带入设备开发中。此外，在不同类型的 Linux 系统中，Python 也可以大大简化应用程序的开发。即便在某些只支持 Java SE 的设备中，Python 依然可以起到简化设计、加速开发的作用。

第 7 章

Python 应用 APP

APP 这个称呼首先来自 iPhone AppStore。但随着移动计算和桌面计算的融合，越来越多的应用程序也归类到 APP。本章主要讨论如何使用 Python 来开发各类桌面应用程序和移动平台 APP/APK，主要关注 APP 开发中用户界面的技术方案和实施细节。

标准化开发是 APP 开发的大趋势。但由于桌面和移动生态平台彼此渗透融合，现有开发平台竞争非常激烈，开发方案的选择反而变得更加复杂了。Android、iOS、Windows、Linux 是几种主流平台。但主流平台内部也存在较大差别。

- 桌面和移动版本的区别：如 Apple 系统中 Mac OS 和 iOS 的差异，以及 Windows 10 桌面和移动设备间的差异。
- GUI 框架区别：操作系统中有时会支持多种 GUI 框架，如 Linux 平台就可同时支持 KDE/GNOME/Xfce/LXDE 等多种 GUI 框架。
- 跨平台框架的融合：Qt/GTK/Java/WebUI 等技术方案可以在不同平台组合上实现不同程度的跨平台运行。

无论对于企业还是开发人员，开发一个 APP 最好能够支持尽可能多的平台，甚至跨越桌面和移动平台就更好了。可惜没有一个可以兼容所有平台的 GUI 框架。不同工具的市场定位不同，兼容性也就不同。这对开发人员，实在不是一件好事情。掌握不同的框架都有学习成本。这些平台不仅框架不同，甚至连开发语言都不同。Android/AWT 使用 Java，Windows 使用.NET 和 C#，iOS 使用 Objective-C，Linux/UNIX 使用 C/C++。不过，我们的确可以实现一定程度上的兼容设计，至少可以将编程语言的数量减少到最少。

最典型的方法就是不使用原生语言，而采用 Python/Java/Swift 的跨平台 GUI 框架。这些框架往往已经将原生 GUI 框架封装起来，简化了开发工作量。此外，利用 HTML5 Web GUI 实现跨平台 APP 也是一种非常流行的方式。

7.1　基于字符的人机界面

现在介绍一下基于字符的命令行和窗口界面。

7.1.1　命令行参数

许多开发者最初都是将 Python 作为脚本或者批处理程序来使用的。命令行往往是最初的人机界面，等应用逐渐复杂了之后再添加 GUI。此外，后台运行的独立程序，如系统服务等，也大多采用命令行参数来启动。

Python 中处理命令行参数的模块主要是 sys、getopt。Python 脚本从操作系统获得的命令行参数保存在 sys.argv 变量中，参数数量则通过 len 函数来获取。

- 参数个数：len(sys.argv)；
- 脚本名称：sys.argv[0]；
- 参数 1：sys.argv[1]，由于 sys.argv[0]是脚本名称，所以脚本参数从 sys.argv[1]开始；
- 参数 2：sys.argv[2]。

```
import sys
import getopt

def usage():
    print "Usage:"
    print "-h, --help\t\t:\tThis help information"
    print "-i, --ip=<addr>\t\t:\tIP address like 192.168.1.101"
    print "-p, --port=<port>\t:\tPort like 8080"

def main():

    try:
        opts, args = getopt.getopt(sys.argv[1:],"hip:",\
            ["help","ip=","port="])
    except getopt.GetoptError, err:
        print str(err)
        usage()
        sys.exit(1)

    for o,a in opts:
        if o == "-v":
            pass
        elif o in ("-h","--help"):
            usage()
            sys.exit()
        elif o in ("-i","--ip"):
            ip = a
        elif o in ("-p","--port"):
```

```
            port = a
        else:
            assert False, "Unhandle options"

if __name__=="__main__":
    main()
```

通过该例子可以清楚地看到 Python 脚本是如何从命令行获取并解析参数的。如果觉得这段代码依然太啰唆，则可以使用 Google 推出的一个 Python 工具以自行分析 Python 脚本并生成命令行接口代码。

7.1.2 字符终端开发

在 GUI/NUI 时代中还要力推字符终端界面，这多少有些反潮流，但对嵌入式系统有一定意义。早期的 Windows 和 Linux 安装界面就是字符终端的界面。Linux Kernel 和某些软件管理包至今还在使用字符界面。从计算机发展史来看，最初的计算机系统都是基于远程终端互联的，人机操作界面都是命令行接口（CLI，Command Line Interface），随后发展出利用 ANSI 控制字符构建的字符终端接口。DEC 的 VT100 至今还是许多终端软件的标配协议。

利用字符构成图像乃至动画是一种计算机艺术形式，图 7-1 基于 Telnet 终端中播放的《星球大战》动画，可让读者了解字符终端做到的极致成果。有兴趣的读者可以使用 TeraTerm 或 Putty 等终端软件，访问 towel.blinkenlights.nl 的 Telnet 端口，观看由字符绘制的动画（它还自带对白）。网站站长 Sten 说，主要的艰苦工作都由 Simon Jansen 完成，他只是将原设计从 Java Applet 迁移到 Telnet 端口而已。

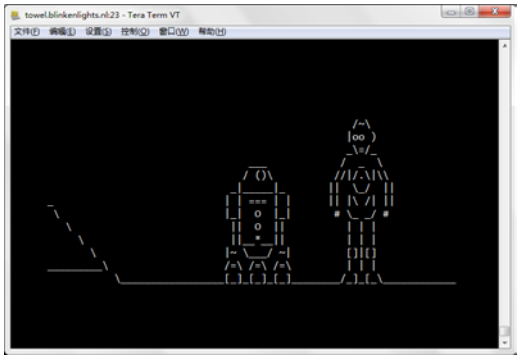


图 7-1 Telnet 端口播放字符构成的《星球大战》动画

终端用户界面不仅仅可以展示简单的动画，在嵌入式开发中还可以节省大量开发人力和时间。

在笔者的资产定位系统设计中，需要利用 PC 通过 USB/UART 对定位设备进行初始化配置和管理，比如写入设备 ID 号、密钥、短消息中心、白名单、黑名单等。原计划编写一个 GUI

客户端，但是由于人力资源受限，笔者改变了开发计划，利用 VT100 协议构建了一个字符型终端。VT100 嵌入在设备固件中，管理员可以用 TeraTerm/Putty/Windows Terminal 对系统进行配置和管理。与 GUI 开发相比，基于终端协议的界面设计的好处如下。

- 跨平台：移动终端和平板电脑中都可以很容易地找到终端软件，所以该设计是跨平台的。
- 适合资源受限平台：对于设备所需要的资源很少，保留 256 字节 RAM 作为用户输入缓冲区即可。
- 开发速度快：由于代码较少，很容易集成在现有代码中。
- 执行速度快：利用快捷键可以在菜单中切换，很适合生产工位设计。
- 兼容设备和控制台接口：不仅管理员可以使用，还有利于主机进行批量自动化配置。

当然，它的局限也有不少：

- 非英语字符串可能出现乱码，这与终端软件的配置有关；
- 不支持与图形图像相关的操作。

所以，字符型终端可以在嵌入式系统和开发时间受限的应用中发挥很大作用。了解其中的字符控制协议还是挺有意义的。之前笔者的项目使用的是 ARM mbed 中的 C++库，系统集成很简单。在深嵌入式系统比如 MicroPython 中，缺乏 curses 库。所以需要重新编写 VT100 的协议库，不过难度不大。

7.1.3　ncurses

如图 7-2 所示，Linux Kernel Menuconfig 的界面，许多读者一定很熟悉，这种界面是基于 Linux 的 ncurses 而开发的。curses 的名字起源于“cursor optimization”，即光标优化。而 ncurses 即“new curses”。其适用于资源有限，没有 GUI 情况下的人机界面，支持表格、菜单、面板、颜色和边框，可以满足基本的用户界面需求。Linux 中著名的 Vi/Vim 就是基于 ncurses 开发的，是服务器端开发者经常使用的文本编辑器。

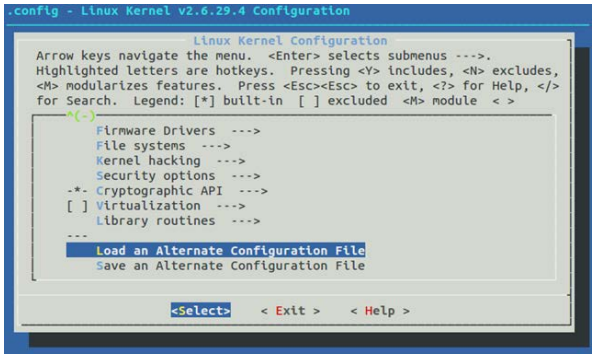


图 7-2　Linux 内核配置界面截图

ncurses 是 UNIX/Linux 下开源的字符终端处理库，也是核心库之一。依赖项为 bash、binutils、coreutils、diffutils、gawk、gcc、glibc、grep、sed 和 make。其主要编程语言是 C 和 Perl。

7.1.3.1 curses

curses 是 Python 在 Linux/UNIX 系统中的 ncurses 封装库。其在 Linux/UNIX 系统中不需要额外安装。以下是一个简单的例子。

```
#!/usr/bin/env python

import curses

myscr = curses.initscr()

myscr.border(0)
myscr.addstr(12,25,"Python curses demo")
myscr.refresh()
myscr.getch()

curses.endwin()
```

7.1.3.2 Windows 文本模式界面

正由于 curses 依赖于 ANSI/POSIX 系统，即 UNIX/Linux，因此如果在 Windows 中导入 curses，则会抛出找不到_curses 库的系统错误。可以在 Windows 中安装 Cygwin/Mingw 后再运行 python curses，或者选用下列推荐的替代模块。

美国加州大学欧文分校的美国国立卫生研究中心的学者 Christoph Gohike，专门为 Windows 下的各类难搞的 Python 扩展包提供了 Python 预编译库，而且针对 32/64 位 Windows 操作系统中不同版本的 Python 2/3 平台分别提供了 Python 软件包的预编译版本。Gohike 提供了对应的 curses 预编译库：http://www.lfd.uci.edu/~gohlke/pythonlibs/# curses

作为一种替代方案，Fredrik Lundh 提供了 Windows 32 下的终端模块，仅支持到 Python 2.6：http://effbot.org/zone/console-index.htm

微软最新的 Windows 10 已经内置 Ubuntu 子系统，支持 Bash shell。经过测试，某导入 curses 模块后可以正常工作。

7.2 桌面 GUI 开发

现阶段虽然存在“移动优先”和“互联网+”的趋势，但是桌面应用在许多情况下依然非常重要。微信作为移动优先的典型案例，也推出了 PC 端应用。以 Web 云服务为基础的 Hotmail、有道笔记、Picasa 等，甚至以自有设备为主的 Kindle 也有 PC 端应用。成熟的产品必然会在多

个平台上进行扩展。尤其对于企业和专业市场的应用程序，桌面开发是必需的。

在物联网相关的应用中，也需要大量的与开发、系统配置、生产相关的桌面 APP。比如：

- Bootloader/ISP/IAP，固件更新；
- EEPROM/NVM，个性化配置和初始化；
- 参数校对，比如传感器的校对、保存等；
- 虚拟仪器和仪表控制板，将物理量以统计图表、实时绘图形式呈现。
- C/S 结构的其他用户软件。

笔者个人并不是 Visual C++/MFC 等传统工具的重度使用者，而只是用其开发过简单的应用程序。主要原因是 Visual 系列开发工具是重型的开发工具，下载、安装、运行都很耗时间。如果仅仅用来开发一些简单应用，过于耗费资源。

Windows 平台上也有一些第三方提供的 GUI 开发工具包，但笔者一直没有留意这些开发平台的发展，毕竟微软才是正统供应商。但随着开源软件的使用日益普及，笔者发现采用此类 GUI 的应用软件越来越多了。而且一旦软件版本升级或发布，往往是针对三大桌面平台发布的，迭代速度很快。典型例子如下。

- GIMP：跨平台的图像处理程序，类似于 Photoshop，基于 GTK+，Python 对应 PyGTK。
- Audacity：跨平台的音频处理程序，基于 wxWidget，Python 对应 wxPython。
- KiCad：跨平台 ECAD 软件，用于电路板设计，基于 wxWidget。
- FreeCAD：跨平台 3D 建模软件，基于 Qt，Python 对应 PyQt。

关于此类例子，请在搜索引擎中检索“Windows 软件替代软件”即可发现长长的清单。这些案例充分说明：不采用原厂工具链也可以开发出兼容性较好的桌面 GUI 和应用程序。开源 GUI 大多来自 Linux/UNIX 或者移动平台，现在则以跨平台开发为目的。这是一种非常值得推荐和尝试的手段，可以大大缩短开发周期，降低开发难度。

以下以 Python 支持的各类 GUI 框架做一些介绍。

7.2.1　Tkinter

Tkinter 模块（Toolkit interface）是 Python 的标准 GUI 工具包接口，是基于 Tcl/Tk 之上的面向对象层封装。Tk 和 Tkinter 可以在大多数的 UNIX 平台下使用，这同样可以应用在 Windows 和 Macintosh 系统里。Tk 8.0 的后续版本可以实现本地窗口风格，即不会在 Windows 系统中很突兀地展现一个 UNIX 外貌风格的程序。Tkinter 可以良好地运行在绝大多数平台中。

Tkinter 依然在不断演变中，所以其软件包的引用需要参考其文档中关于版本说明的细节。最简单的例子如下：

```
import Tkinter
top = Tkinter.Tk()
label = Tkinter.Label(top, text='Hello World')
```

```
label.pack()
Tkinter.mainloop()
```

推荐 Tkinter 用于实现一些相对简单的应用程序。因为 Tkinter 是一种基本的 UI 工具箱，相对于 wxPython 和 PyQt UI 框架，Tkinter 在完整度和可视化开发工具上稍逊一筹。

如果读者有足够耐心的话，建议阅读 Tkinter 专题教程书籍：*Tkinter 8.4 Reference - Python GUI*。

7.2.2 wxPython

介绍 wxPython 必须提到 wxWidgets。wxWidgets 是一款优秀的开源 GUI 图形库，具备非常优秀的跨平台能力。wxWidgets 最初的名字是 wxWindows，2004 年其受到微软抗议，改名为 wxWidgets。wxWidgets 采用 C++编写，wxPython 是它的 Python 封装，并以 Python 模块的方式提供给用户使用。

wxPython 是一套优秀的 Python GUI 图形库，允许 Python 程序员方便地创建完整、功能健全的 GUI 用户界面，并成为 Python UI 的事实标准之一。与 Python 和 wxWidgets 一样，wxPython 也是一款开源软件，能够运行在 32/64 位 Windows、绝大多数的 UNIX 或类 UNIX 系统、Macintosh OS X 中。

7.2.2.1 wxPython 安装

wxPython 的安装不适用于 pip，Windows/Mac 版本需要自行下载。下载地址如下：http://www.wxpython.org/download.php

其中 Windows 的版本有 32/64 位版本，该版本与 Python 环境的版本有关。如果你的 Python 是 32 位，那么即使操作系统是 64 位，也请下载 32 位版本。

Linux（Ubuntu/Debian）的安装指令为：

```
sudo apt-get install python-wxgtk2.8 python-wxtools wx2.8-i18n
```

或者参考以下网址进行配置：http://wiki.wxpython.org/InstallingOnUbuntuOrDebian。

7.2.2.2 wxPython 例子

由于使用 Python 作为编程语言，因此与 C++实现相比，wxPython 编写简单、易于理解。示例如下：

```
import wx
class App(wx.App):
    def OnInit(self):
        frame=wx.Frame(parent=None,title='MyFirstWxPythonApplication')
        frame.Show()
        return True
app=App()
```

```
app.MainLoop()
```

这个基本的 wxPython 程序说明了开发 wxPython 程序所必需的五个基本步骤：

（1）导入 wxPython 包；

（2）继承 wxPython 应用程序类；

（3）定义应用程序的初始化方法；

（4）创建应用程序类的实例；

（5）进入这个应用程序的主事件循环。

读者设计 GUI 时，可以先尝试用 wxPython 做些测试和学习。在实际工程中，可以使用 GUI 构建器来简化 GUI 开发。

7.2.3　Boa Constructor

Boa Constructor 是一个跨平台的 Python 集成开发环境和 wxPython 图形用户界面构建器。它提供了可视化方式的框架（即窗口） 创建和处理、对象检视器（object inspector）、编辑器、继承等级、HTML 文档字符串、高级调试器和集成化帮助系统，其与主流的可视化工具类似。其软件截图请参见图 7-3。

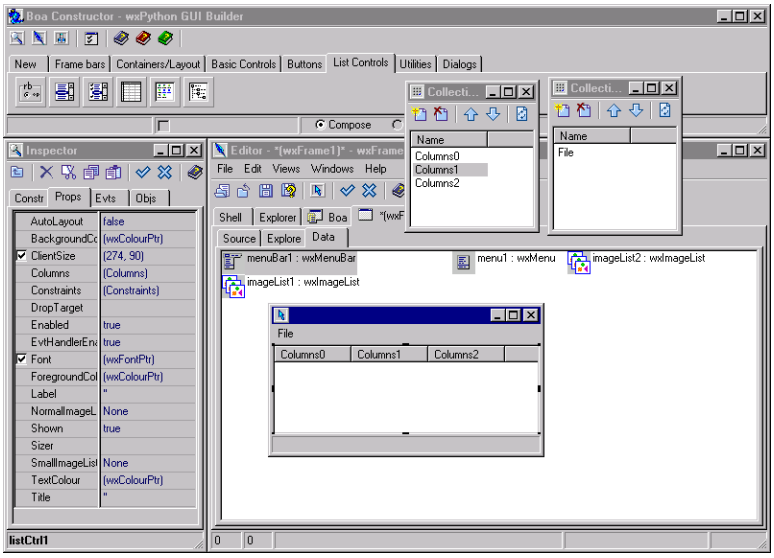


图 7-3　Boa Constructor IDE 截图

对于初学者或者对程序结构要求不高的用户来说，使用 Boa Constructor 这样的基于 wxPython 的开发平台，可以相对轻松地开发出优秀的 wxPython 程序。

7.2.4 wxGlade

wxGlade 和 Boa Constructor 一样，是针对 wxWidgets/wxPython 的 GUI 设计工具，它像 Visual Basic 里面的设计工具一样。它可以产生 Python（wxPython）、C++/Perl/Lisp（wxWidgets）、XRC（wxWidgets 的 XML 资源文件）代码。此外，wxGlade 本身也是由 Python 编写的。其软件界面可参见图 7-4。

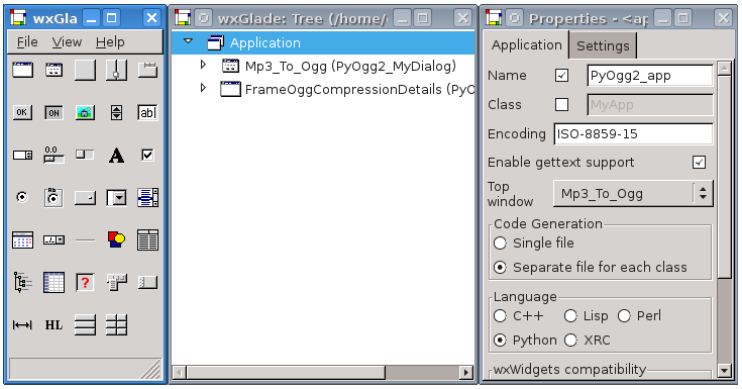


图 7-4 wxGlade 软件截图

wxGlade 与 Boa Constructor 的不同在于：wxGlade 仅仅是 GUI 设计软件，并非完整的 IDE；且其不仅仅产生 Python 代码，它还针对 GTK+/GNOME 的其他开发语言，开发者可能需要配合其他语言来编写业务代码。

许多 UI 系统都使用 XML 格式作为 GUI 资源描述文件。这种技术趋势可以将资源描述与 GUI 框架和业务逻辑分离，满足 GUI 的面向对象需求。该技术广泛用于多种框架和平台。Android UI 也使用 XML 进行描述。XRC 是 wxGlade 的 XML 资源文件，wxGlade 提供了一个 XRC 样本文件：http://wxglade.sourceforge.net/samples/ppp_connection.xrc。

Python 是一种易于使用的语言，但是 GUI 开发代码编写还是很繁重的。如果在设计阶段借助 Boa 或 wxGlade 来产生基础 Python 代码，之后添加业务相关的 Python 代码，则可以充分发挥出 Python 快速原型开发的高效率。

与 wxPython 类似，wxGlade 需要单独下载，不支持 pip。网址如下：https://sourceforge.net/projects/wxglade/

除了 Boa 和 wxGlade，类似的构建工具如下：PythonCard、XRCed、VisualWx。读者可以自行下载评估，在此不再一一赘述。

7.2.5　PyGTK

GTK+，是用 C 语言开发的跨平台的 GUI 库，它是 Linux 的主流桌面系统 GNOME 桌面系统和 GIMP 图像编辑器的开发工具箱，是世界上许多程序员的首选 GUI 框架。

GTK+提供了各式可视元素和功能。PyGTK 可让用户使用 Python 轻松创建 GTK 图形用户界面程序。

PyGTK 具有跨平台性，它能不加修改地稳定运行在各种操作系统如 Linux、Windows、Mac OS 之上。除了具有简单易用和快速原型开发能力外，PyGTK 还有一流的本地化语言处理能力。PyGTK 是自由软件，它是基于 LGPL 协议发布的。开发者可以自由使用、修改、分发和研究。

PyGTK 安装与依赖项目：

- 32 位 Python 解释器；
- GTK+运行时环境，32/64 位均可；
- PyGTK 依赖于 PyCairo/PyGObject 包。

也可下载官网的集成安装包简化安装过程。PyGTK 默认安装的是 Windows 32 位系统，所以 Windows 7/8/10 系统有两个选择：

- 从 Python 解释器开始到 PyGTK 及其依赖项目安装，一律使用 32 位配置；
- 下载 7.1.3.2 节中提到的 gohike 提供的 64 位 PyGTK 预编译库。

PyGTK 可以使用 GtkBuilder 和 Glade 两种方式构建 UI。Glade 是专为 GTK+和 GNOME 开发的用户界面构建器，用于产生描述 UI 的 XML 文件。

helloworld.py：

```
#!/usr/bin/env python

# example helloworld.py

import pygtk
pygtk.require('2.0')
import gtk

class HelloWorld:

    # This is a callback function. The data arguments are ignored
    # in this example. More on callbacks below.
    def hello(self, widget, data=None):
        print "Hello World"

    def delete_event(self, widget, event, data=None):
        # If you return FALSE in the "delete_event" signal handler,
        # GTK will emit the "destroy" signal. Returning TRUE means
        # you don't want the window to be destroyed.
```

```
        # This is useful for popping up 'are you sure you want to quit?'
        # type dialogs.
        print "delete event occurred"

        # Change FALSE to TRUE and the main window will not be destroyed
        # with a "delete_event".
        return False

    def destroy(self, widget, data=None):
        print "destroy signal occurred"
        gtk.main_quit()

    def __init__(self):
        # create a new window
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)

        # When the window is given the "delete_event" signal (this is given
        # by the window manager, usually by the "close" option, or on the
        # titlebar), we ask it to call the delete_event () function
        # as defined above. The data passed to the callback
        # function is NULL and is ignored in the callback function.
        self.window.connect("delete_event", self.delete_event)

        # Here we connect the "destroy" event to a signal handler.
        # This event occurs when we call gtk_widget_destroy() on the window,
        # or if we return FALSE in the "delete_event" callback.
        self.window.connect("destroy", self.destroy)

        # Sets the border width of the window.
        self.window.set_border_width(10)

        # Creates a new button with the label "Hello World".
        self.button = gtk.Button("Hello World")

        # When the button receives the "clicked" signal, it will call the
        # function hello() passing it None as its argument.  The hello()
        # function is defined above.
        self.button.connect("clicked", self.hello, None)

        # This will cause the window to be destroyed by calling
        # gtk_widget_destroy(window) when "clicked".  Again, the destroy
        # signal could come from here, or the window manager.
        self.button.connect_object("clicked", gtk.Widget.destroy, self.window)

        # This packs the button into the window (a GTK container).
        self.window.add(self.button)

        # The final step is to display this newly created widget.
```

```
        self.button.show()

        # and the window
        self.window.show()

    def main(self):
        # All PyGTK applications must have a gtk.main(). Control ends here
        # and waits for an event to occur (like a key press or mouse event).
        gtk.main()

# If the program is run directly or passed as an argument to the python
# interpreter then create a HelloWorld instance and show it
if __name__ == "__main__":
    hello = HelloWorld()
    hello.main()
```

7.2.6　PyQt

Qt 是奇趣科技于 1991 年推出的跨平台 C++图形用户界面应用程序开发框架，其徽标如图 7-5 所示。它既可以开发 GUI 程序，也可用于开发非 GUI 程序，比如控制台工具和服务器。Qt 是面向对象的框架，使用特殊的代码生成扩展以及宏，易于扩展，允许组件编程。2008 年，奇趣科技被 Nokia（诺基亚）公司收购，Qt 也因此成为 Nokia 旗下的编程语言工具。2012 年，Qt 被 Digia 收购。2014 年 4 月，跨平台集成开发环境 Qt Creator 3.1.0 正式发布，至此其实现了全面支持 iOS、Android、WP。

图 7-5　奇趣公司 Qt 徽标

Qt 提供了开发图形用户界面所需的所有功能。Qt 很容易扩展，并且允许真正的组件化编程。基本上，Qt 同 X Window 上的 Motif、Openwin、GTK 等图形界面库及 Windows 平台上的 MFC、OWL、VCL、ATL 是同类型的东西。

PyQt 是一个创建 GUI 应用程序的工具包。它是 Python 编程语言和 Qt 库的成功融合，由 Phil Thompson 开发。PyQt 实现了一个 Python 模块集。PyQt 有 300 多个类，以及将近 6000 个函数和方法。作为一个多平台工具包，其可以运行在所有主要操作系统上，包括 UNIX、Windows 和 Mac。

因为可用的类有很多，所以其可以分成几个模块。

- QtCore 模块包含核心的非 GUI 功能，用于时间、文件和目录、各种数据类型、流、网

址、MIME 类型、线程或进程。

- QtGui 模块包含图形组件和相关类，例如按钮、窗体、状态栏、工具栏、滚动条、位图、颜色、字体等。
- QtNetwork 模块包含了网络编程类，其用于编写 TCP/IP 和 UDP 的客户端和服务器，使网络编程更简单、更轻便。
- QtXml 模块提供了 SAX 和 DOM API 的实现。
- QtSvg 模块提供了显示 SVG 文件的类。可缩放矢量图形（SVG）是一种用于描述二维图形和图形应用程序的 XML 语言。
- QtOpenGL 模块使用 OpenGL 库渲染 3D 和 2D 图形，能够无缝集成 Qt 的 GUI 库和 OpenGL 库。
- QtSql 模块，用于数据库访问。

PyQt 所依赖的 Qt4/Qt5 是移动设备中最常见的 C++ GUI。PyQt 支持商业开发，在技术支持和软件完整度方面比较优秀。

7.2.6.1 PyQt 安装

开发者可以到 sourceforge PyQt 站点下载对应的安装包。

PyQt 采用双许可证，开发人员可以选择 GPL 和商业许可。在此之前，GPL 的版本只能用在 UNIX 上，从 PyQt 的版本 4 开始，GPL 许可证可用于所有支持的平台。不过建议利用 PyQt 开发商业软件依然要仔细阅读其软件许可证，或者使用 PySide 替代 PyQt。

7.2.6.2 Eric Qt GUI 编辑器

Eric 是一个完整的 Python 编辑器和 IDE，采用 Python 编写。Eric 本身采用 Qt 开发，集成了 Scintilla 编辑器。可以将其作为日常的编辑器、专业的项目管理工具。其本身采用插件开发。采用 Eric 可以非常方便地开发 PyQt 应用程序。

Eric 下载网址：http://eric-ide.python-projects.org/。

Eric 依赖于 Python/Qt/PyQt/QScintilla，还可以选装：Mercurial、Subversion、PySvn、Git、PyLint、CX_Freeze、PyEnchant。其可用于软件配置管理、代码美化和可执行文件转换。

7.2.7 PySide

PySide 是跨平台的 Qt GUI 的 Python 绑定，由 Nokia/Qt 团队开发，其徽标如图 7-6 所示。在 Qt 被并入 Nokia 的日子里，奇趣公司意识到 Qt 的 GPL 协议对于商业开发的限制，单独推出了 PySide。该软件包提供了和 PyQt 类似的功能，API 也兼容 PyQt，采用 LGPL 许可证，可以用于商业软件开发。

图 7-6　PySide 徽标

安装：

```
pip install PySide
#!/usr/bin/python
import sys
from PySide.QtCore import *
from PySide.QtGui import *

# Create a Qt application
app = QApplication(sys.argv)

# Create a Label and show it
label = QLabel("Hello World")
label.show()

# Enter Qt application main loop
app.exec_()
sys.exit()
```

Qt/PyQt/PySide 是非常值得推荐的跨平台 GUI 组合。

7.2.8　Enthought

Enthought 是一家专注于科学计算的公司，围绕科学计算提供了一大批开源组件，如 SciPy 等。在以后的章节中再继续介绍该公司推出的其他产品。Enthought 提供的 ETS（Enthought Tool Suites）不仅仅包括 TraitsUI 框架，还有一些应用相关框架。比如插件管理等，可以用于构建一个复杂的应用程序。所以，它提供的解决方案既可以解决应用的快速开发问题，也可以用来构建更加高级的、支持扩展插件机制的可扩展应用框架。读者可以使用这些开源组件来开发自己的平台软件，例如为 MicroPython 开发一个类似于 Arduino/Processing 的 IDE。

7.2.8.1　Traits 与 TraitsUI

Traits 的词根是 trait，复数为 traits，英文原意：特征。作为 Enthought 的软件产品，其首字母为大写。

Python Traits 库分为两部分：Traits 和 TraitsUI。Traits 为 Python 添加了类型定义的功能，使其具备初始化、校验、代理、事件等诸多功能。TraitsUI 基于 MVC（Model-View-Controller）

模型构建人机界面。MVC 常见于 Web 应用和 UI 设计。在 Traits UI 的开发中，可以选择 wxPython 和 PyQt 两种软件包作为底层实现。

7.2.8.2 Traits 赋予对象附加属性

Python 作为一种动态类型编程语言，它的变量类型可变，没有固定类型，这种灵活性给快速开发带来了很多便利，不过它也不是没有缺点的。Python Traits 库的一个很重要的目的就是为了解决这些缺点所带来的问题。

Traits 库可以赋予对象属性以下一些附加的特性。

- 初始化（Initialization）：trait 有默认值，在程序中初次使用之前，该属性就被自动赋予一个初始值。
- 验证（Validation）：trait 属性的类型是显性声明的。该类型在用户代码中可见，而且必须符合开发者设定的类型组合才可以赋值给这些属性。
- 代表（Delegation）：trait 属性的值可以包含在定义的对象中，也可以在 trait 代表的另一对象中。
- 通知（Notification）：设置 trait 属性数值，可以将这种变化通知到程序的其他部分。
- 可视化（Visualization）：用户界面允许用户交互式地修改 trait 属性，并使用 trait 定义进行自动构建。此项功能需要底层 GUI 支持。

一个类可以混用 trait 属性与 Python 属性，在类中定义的 trait 属性可以自动被该类的任意子类继承。

Enthought 引入 Python Traits 库最初是为了开发旗下 Chaco 2D 绘图库而设计的。该绘图库中有很多绘图用的对象，每个对象都有很多诸如线型、颜色、字体之类的属性。为了方便用户使用，每个属性可以允许多种形式的值。

7.2.8.3 Traits 快速 UI 开发

Python 有着丰富的界面开发库，除了默认安装的 Tkinter 以外，wxPython、PyQt4 等都是非常优秀的界面开发库。但是它们有一个共同的问题：需要开发者掌握数量众多的 API 函数。许多细节，例如配置控件的属性、位置以及事件响应都需要开发者一一处理。虽然 wxPython/PyQt 也都提供了一些开发工具，但依然显得过于烦琐。

在开发程序时，我们希望快速实现一个能用的界面，让用户能够交互式地处理数据，而不希望在界面制作上花费过多的精力。以 Traits 为基础、以 MVC 为设计思想的 TraitUI 库就是实现这一理想的最佳伴侣。而 Traits/TraitsUI 可以帮助开发者简化处理控件细节。

安装：

```
pip install traits
pip install traitsui
```

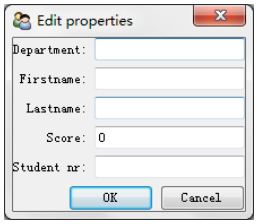
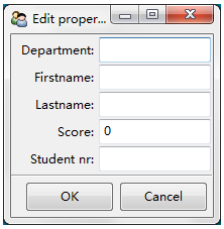
两个软件包加起来大约 10MB。

示例如下：

```
#!/usr/bin/env python

from traits.api import HasTraits, Str, Int

class SimpleStudent(HasTraits):
    firstname = Str
    lastname = Str
    department = Str
    student_nr = Str
    score = Int

allan = SimpleStudent()
allan.configure_traits()
```

TraitsUI 可以选择不同的 GUI 后端：Qt4 和 wxPython，两者运行效果如表 7-1 所示。在 Qt4 版本运行过程中可能报警提示 msvcr120.dll 缺失，但即使缺乏该 DLL 文件，UI 最终也会出现。如果要避免此类报警，需要安装 Microsoft VC++ runtime redistribution 2013，在微软官网下载 vcredist_x64.exe 或 vcredist_x86.exe。wxPython 版本运行无问题，但是结束后会出现某些报警：如关闭一个未经初始化的模块等。这应该和代码初始化有一定关联。安装 Enthought 的完整安装包可以避免出现上述各类报警信息。

表 7-1　TraitsUI 采用不同 GUI 后端效果对比

GUI 后端	Qt4	wxPython
命令	traitsDemo.py -toolkit qt4	traitsDemo.py -toolkit wx
截图		

7.2.9　Cocoa+PyObjC

针对 Mac 桌面，我们可以使用 Cocoa+PyObjC。Mac 桌面并非笔者关注的重点，请读者自行查看对应的文档了解详情。在 Mac 平台中崛起的 Swift 语言值得大家注意。当前，Swift 已经在 Windows、Linux 和 Mac，以及 iOS 和 Android 中得到实现，成为一种新的跨平台开源编程语言。

7.2.10 Java AWT

历史悠久的 Java J2SE 自带 AWT 和 JavaFX 两种 GUI，基于 JVM 的 Jython 自然而然地可以充分利用 JVM 的跨平台特性编写桌面应用。更加重要的是采用 Jython 编写代码的工作量要少于 Java 原生代码。

Jython AWT GUI

Java AWT 演示代码如下：

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class HelloWorld {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello Java!");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        JButton button = new JButton("Click Me!");
        button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    System.out.println("Clicked!");
                }
            }
        );
        frame.add(button);
        frame.setVisible(true);
    }
}
```

Jython 代码如下：

```
from javax.swing import JButton, JFrame

frame = JFrame('Hello, Jython!',
           defaultCloseOperation = JFrame.EXIT_ON_CLOSE,
           size = (300, 300)
       )

def change_text(event):
    print 'Clicked!'

button = JButton('Click Me!', actionPerformed=change_text)
frame.add(button)
```

```
frame.visible = True
```

由于 Jython 实际上运行在 Java VM 之上，所以其运行结果与 Java AWT 是一样的。运行效果如图 7-7 所示。

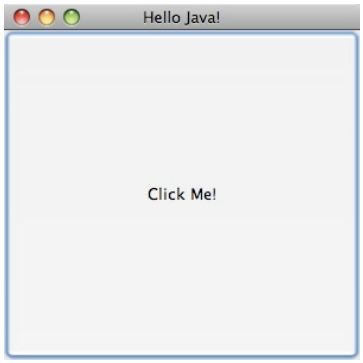


图 7-7　Jython GUI 运行效果

具体的 Jython GUI 编程可查看其官网教程：

http://www.jython.org/jythonbook/en/1.0/GUIApplications.html

7.2.11　IronPython 与 WPF

WPF（Windows Presentation Foundation）是微软推出的基于 Windows Vista 的用户界面框架，其属于.NET Framework 3.0 的一部分，可以使用 IronPython 进行编程。这为开发者增加了一种选择。

7.2.12　其他 UI

此外，Python 还有大量的相对小众的 GUI 库。现罗列如下：

- pyui4win，国产开源软件，基于 duilib，有界面生成器。
- urwid，创建终端 GUI 应用的库，支持组件、事件和丰富色彩。
- AutoPy，较为简单的跨平台 Python GUI 工具包。使用纯 ANSI C 编写而成，可运行在 Mac OS X、Windows 和 X11（UNIX/Linux）上。
- pywin32，采用 VC 的形式来使用 Python 开发 Win32 应用。代码风格可以类似于 Win32 SDK，也可以类似于 MFC。
- Toga，支持操作系统原生 GUI。
- Tix，Tk interface extension 是一个强大的 Python UI 组件。使用 Tix 可大大改善图形界面应用的显示效果和功能。

- Dabo，wxPython 的再封装库。它提供了数据库访问、商业逻辑以及用户界面。
- Gaphas，应用于 GTK+程序之上的图形控件，采用 Cairo 渲染的 MVC 容器。
- Deepin UI，针对应用程序需求而封装的上层 UI 库，底层依赖于 GTK+。其支持主题切换、自定义控件外观和模块。
- pyglet，Python 跨平台窗口及多媒体库。
- Python-SIP，生成 C++接口代码的工具，它与 SWIG 类似，但使用不同的接口格式。它用作创建 PyQt 和 PyKDE 应用，并支持 Qt signal/slot 系统。
- Camelot，基于 Sqlalchemy 和 PyQt 开发的 Python 的 GUI 框架，用来构建桌面图形化界面的应用。
- PyMT，Python 多点触摸用户界面库。
- PFramer，QFramer 的 Python 增强版本。其兼容 PySide/PyQt4/PyQt5 的各个版本，兼容 Python 2.7 和 Python 3.4，具备完整的 QSS 换肤机制。
- PyGUI，面向 UNIX、Mac 和 Windows 平台。作为 MVC 框架，PyGUI 的开发理念是能够更好、更容易地融入 Python 生态系统。

开源软件的特点就是多，目不暇接，仅仅评估这些软件就需要很多的时间。

7.3 本地 Web GUI

早期联网应用开发中经常听到的术语有 C/S（Client/Sever）和 B/S（Browser/Server）模式，还有 RIA（Rich Internet Application）应用。渐渐地，开发者发现利用不同的 UI 系统进行开发是件成本很高的事情。如果利用同一种技术去开发不同平台的 UI 则是一件节省成本的方法。所以，越来越多的开发者利用 Web 前端技术组合（HTML5+JavaScript+CSS）来开发，至于后端业务逻辑是放在本地还是远程则根本不重要。Web GUI 的迭代速度快、可以实现持续交付，这也是互联网服务保持吸引力的重要手段。

基于本地 Web 服务器，利用 HTML5+JavaScript 做 GUI 的方式可以跨越 Web 和桌面，这已经成为一种技术趋势。从这个趋势看，JavaScript 不仅会跨越前端、服务器后端，也会跨越 Web/Desktop 甚至嵌入式，而成为一种全栈开发语言。虽然使用 JavaScript 开发 Web GUI 更加合理，但 Python 也可以用于构建 Web GUI 应用程序。

在物联网领域，网络编程成为常态，所以采用 Web GUI 有很大应用价值。接下来介绍几种 Web 应用开发框架。

7.3.1　与 WebKit 相关的 Python 包

Apple WebKit 是现在主流的浏览器引擎之一。iOS/Mac OS 中的 Safari、Android 浏览器/Chrome 浏览器等都是基于 WebKit 制作的浏览器。Python WebUI 基于 WebKit 是非常自然的事情。在 Python 中有多个 Webkit 的绑定：

- wxWebKit，WebKit 渲染引擎的 wxWidgets 实现。其支持 Windows 和 Mac，但却很久没有得到更新了。
- PythonWebKit，提供对于 WebKit 引擎的直接 Python DOM 绑定。它将 Python 置于 JavaScript 对等的地位上。开发者可以在 Python 中直接使用 getElementsByTagName、appendChild 等函数，甚至将 Python 的事件回调注册到 XMLHttpRequest、Window 和 HTML 元素上。
- PyWebKitGTK，提供 Webkit 的 Python 绑定，并针对 GTK2 编译。最近的进展是提供完整的 DOM 操纵能力，包括 JavaScript 代码执行和运算。PyWebKitGTK 是 PyjamasDesktop 的基础。
- PyWebKitQt4，PyQt 对于 WebKit 的绑定。

利用 WebKit Python 封装包，再配合本地 Web Server，可以相对简单地构建本地 UI。

7.3.2　OneRing

OneRing 是豆瓣网开发的开源跨平台桌面应用框架。和 Adobe AIR 类似，它支持用 HTML/JS/CSS 制作用户界面；与之不同的是，它的应用为本地程序，可以直接访问操作系统的数据。豆瓣 FM 就是依托 OneRing 框架搭建的。

很可惜的是，它的代码托管在 Google code，网址如下：

http://code.google.com/p/onering-desktop/

OneRing 基本上是依托于 Qt WebView 做的框架，WebView 本身是 WebKit 封装，具备了跨平台能力。OneRing 另一个比较大的特点是对多种语言的绑定，源码中有 Python 绑定的实例。

其整个开发流程还挺复杂的，需要安装 C 编译器、Qt，生成 makefile，再编译成 DLL 库，然后再构建 Python 绑定，并安装 OneRing。

7.3.3　Pyjs

Pyjs 是用于 Web 和桌面应用开发的 RIA 开发平台，包含 Python/JavaScript 编译器、AJAX 框架和 Widget API。Pyjs 相当于谷歌的 GWT 的 Python 版本，同时也支持桌面应用开发：Pyjs Desktop。可直接使用独立桌面应用程序的方式来运行相同代码的 Web 项目，而无须打开浏览器。

Pyjs 官网中有不少实例，但笔者观察下来发现，其有一段时间没有更新了。其是否适用于

工程项目需要读者自行评估。此外，Windows 平台中使用的技术与 Linux 略有不同。

7.3.4 Python Flexx

Flexx 采用纯 Python 编写，用于创建 GUI 程序的工具集，并使用 Web 技术进行界面渲染。笔者测试下来发现，Flexx 依赖于 Tornado，而且其底层实现是基于 XUL 的。XUL 是一种基于 XML 的语言。XUL 建立在 Web 技术之上：HTML、JavaScript 和 CSS。如果想有效地使用 XUL，需要对这些技术，特别是 XML 名称空间非常熟悉。

Flexx 使用模块化设计，包含如下一些子系统：

- ui——UI 部件；
- app——事件循环和服务器；
- react——reactive 编程；
- pyscript——Python/JavaScript 转换编译器；
- webruntime——启动运行时。

安装：

```
sudo pip install tornado flexx
```

flexx_demo.py:

```
from flexx import app, ui, react

class Example(ui.Widget):

    def init(self):
        self.count = 0
        with ui.HBox():
            self.button = ui.Button(text='Click me', flex=0)
            self.label = ui.Label(flex=1)

    @react.connect('button.mouse_down')
    def _handle_click(self, down):
        if down:
            self.count += 1
            self.label.text('clicked %i times' % self.count)

main = app.launch(Example)
app.run()
```

运行效果如图 7-8 所示。

图 7-8　Flexx 运行效果

Flexx 目前还处于开发阶段，演示代码也比较简陋，文档不算特别完整。

7.4　本地可执行文件

在大部分情况下，Python 以源码形式交付，而商业化程序需要打包成本地可执行文件。具体的做法就是将 Python 解释器变成一个 DLL（或者其他形式的）库，并把 Python 程序和所依赖的 Python 模块、库和扩展编译成 pyc/pyo/DLL 格式，一起复制到目标文件夹中。本节就介绍一些打包工具。

7.4.1　Linux 可执行文件

Linux 可执行文件的概念和 Windows 不同，与文件类型无关，而与文件权限有关。

在 Linux 中，不仅 GCC 编译后的二进制文件可以是可执行文件，文本文件如 shell 脚本、Python/Perl/PHP/Lua 等也可以是可执行文件。使用 ls -al 可以查阅各个文件的读、写、执行权限。

```
root@ubuntu-vm:~/rq_lab# ls -al
total 16
drwxr-xr-x  2 root root 4096 May 16 22:18 .
drwx------ 21 root root 4096 May 23 10:59 ..
-rwxr-xr-x  1 root root  110 May 16 22:15 demo.py
-rw-r--r--  1 root root  241 May 16 22:18 rqtest.py
```

rwx 分别代表读/写和执行权限。在目录中，demo.py 具备可执行权限，rqtest.py 不具备可执行权限。

Python 在 Linux 下并不需要将.py 转换为.pyc/.pyo 文件也可以执行，但需要在 Python 源文件头部写上 shebang（即“#!”组合）字符串。

```
#!/usr/bin/env python
```

当然，必须将其设定为可执行的权限。

```
chmod 0755 yourscript.py
```

因为 Python 是完整版 Linux 标准组件之一。任何 Python 源码和编译后的文件都可以在 Python 运行时环境中直接执行。

7.4.2 Mac OS X 应用程序包

Mac OS X 来自 UNIX。Python 源码也可以直接运行。Mac OS X 应用程序后缀为 dmg。

7.4.3 Windows 可执行文件

在 Windows 中采用 exe 作为可执行文件后缀，那么用 Python 编写的应用如何发布给客户呢？可以使用打包工具将 Python 打包为 exe 文件。

7.4.4 pyinstaller

pyinstaller 是一个跨平台的打包工具，支持以下操作系统：

- Windows 32/64 位；
- Linux 32/64 位；
- Mac OS X 32/64 位；
- Solaris/AIX UNIX，实验性功能。

笔者使用后发现，pyinstaller 的兼容性比较好。笔者个人比较常用的是 pyinstaller。除了下载安装包，使用 pip 也可以安装 pyinstaller。其依赖项是 pefile、pypiwin32 和 future。

```
pip install pyinstaller
```

使用 pyinstaller 时，可以将程序打包成终端（命令行）或者窗口类型。在打包 GUI 应用程序时，记得要关闭终端选项，否则运行时会闪现一个黑黑的终端窗口。

7.4.5 py2exe

py2exe 是一个将 Python 脚本转换成 Windows 可执行程序的打包工具。这样在客户系统中即使没有 Python 环境也可以运行 Python 脚本。py2exe 支持的平台主要是 Win32 平台，所以其很适合 Windows XP 等 32 位操作系统。

7.4.6 py2app

py2app 是一个跨平台的安装工具包，可以将 Python 脚本打包成 Windows 可执行文件或 Mac OS X 的应用程序包。该项目依赖于 altgraph、modulegraph、macholib。

安装：

```
pip install py2app
```

7.4.7　cx_Freeze

cx_Freeze 是一组脚本和模块，可将 Python 脚本转换到可执行文件，其非常类似于 py2exe/py2app。和这两者不同的是，cx_Freeze 是跨平台的，其可以在 Python 支持的任意平台上使用。

安装：

```
pip install cx_Freeze
```

安装包的名称可以是 cx_Freeze，也可以是 cx-Freeze。

读者可以根据自己的目标平台操作系统类型和版本选择 py2exe/pyinstaller/cx_Freeze。

7.4.8　Windows 系统服务

Python 程序一般作为用户程序运行，如果要实现后台运行，必须将其转化为系统服务。利用 Windows 自带的 sc.exe 可以将 exe 转换成后台运行的系统服务。sc.exe 是用于与服务控制管理器和服务进行通信的命令行程序。如果将 Python 脚本封装成 exe，再配合 SC，就可以构建自己的系统服务程序。一些物联网模块的守护程序可以通过这种方式构建。

```
c:\>sc
DESCRIPTION:
        SC is a command line program used for communicating with the
        NT Service Controller and services.
USAGE:
        sc <server> [command] [service name] <option1> <option2>...

        The option <server> has the form "\\ServerName"
        Further help on commands can be obtained by typing: "sc [command]"
        Commands:
          query-----------Queries the status for a service, or
                          enumerates the status for types of services.
          queryex---------Queries the extended status for a service, or
                          enumerates the status for types of services.
          start-----------Starts a service.
          pause-----------Sends a PAUSE control request to a service.
          interrogate-----Sends an INTERROGATE control request to a service.
          continue--------Sends a CONTINUE control request to a service.
          stop------------Sends a STOP request to a service.
          config----------Changes the configuration of a service (persistant).
          description-----Changes the description of a service.
          failure---------Changes the actions taken by a service upon failure.
          sidtype---------Changes the service SID type of a service.
          qc--------------Queries the configuration information for a service.
          qdescription----Queries the description for a service.
          qfailure--------Queries the actions taken by a service upon failure.
          qsidtype--------Queries the service SID type of a service.
```

```
          delete----------Deletes a service (from the registry).
          create----------Creates a service. (adds it to the registry).
          control---------Sends a control to a service.
          sdshow----------Displays a service's security descriptor.
          sdset-----------Sets a service's security descriptor.
          showsid---------Displays the service SID string to an arbitrary name.
          GetDisplayName--Gets the DisplayName for a service.
          GetKeyName------Gets the ServiceKeyName for a service.
          EnumDepend------Enumerates Service Dependencies.

        The following commands don't require a service name:
        sc <server> <command> <option>
          boot------------(ok | bad) Indicates whether the last boot should
                          be saved as the last-known-good boot configuration
          Lock------------Locks the Service Database
          QueryLock-------Queries the LockStatus for the SCManager Database
EXAMPLE:
        sc start MyService

Would you like to see help for the QUERY and QUERYEX commands? [ y | n ]:
```

7.4.9 Windows 定时任务

具体示例如下：

```
C:\>schtasks

任务名                                 下次运行时间             状态
====================================== ======================= ===============
微软设备健康助手自动更新               12:19:00, 2016-6-20
```

schtasks 的运行参数：

```
C:\>schtasks /?

SCHTASKS /parameter [arguments]

描述:
    允许管理员创建、删除、查询、更改、运行和中止。
    本地或远程系统上的计划系统。替代 AT.exe。

参数列表:
    /Create         创建新计划任务。
    /Delete         删除计划任务。
    /Query          显示所有计划任务。
    /Change         更改计划任务属性。
```

```
/Run            立即运行计划任务。
/End            中止当前正在运行的计划任务。
/?              显示帮助/用法。
```

示例：

```
SCHTASKS
SCHTASKS /?
SCHTASKS /Run /?
SCHTASKS /End /?
SCHTASKS /Create /?
SCHTASKS /Delete /?
SCHTASKS /Query  /?
SCHTASKS /Change /?
```

在 Windows 中，schtasks 用于安排命令或者程序定期运行或在指定时间运行，类似于 Linux 中的 crontab。Windows 中的 sc 是用来与服务程序通信的命令行程序，sc 所调用的脚本本身是服务程序，服务程序中也可以实现类似于 schtasks 的功能，但是需要自行管理进程的启动、中止和定时功能。

7.4.10　Linux 系统服务

由于 Linux/UNIX 的许多系统服务都以套接字形式提供，所以不得不提到另一个超级服务器：inetd。它是监视网络请求的守护程序，并根据请求来调用相应服务程序处理连接请求；它同时也有能力处理一些简单的请求。

在类 UNIX 操作系统中，使用 init 来产生所有其他进程的程序，以守护进程方式存在，init 的 pid 为 1。其中，最常见的就是 System V 的 init 程序。init 将需要初始化的程序逐个串行运行，代价是加载速度慢，所以出现了许多替代程序。

大部分 Linux 发行版的 init 与 System V 风格兼容，但是 Arch/Slackware/FreeBSD/NetBSD 则采用 BSD 风格，Gentoo 采用定制 init，Ubuntu 从 6.10 版开始使用 upstart 来替代 init，这些 init 替代品均可以实现一定程度上的并行异步启动。但 Ubuntu 从 15.04 版开始从 upstart 转向 systemd。

systemd 是一个大一统的管理程序，它不仅仅替代了 sysvinit，还替代了 pm-utils、inetd、acpid、syslog、watchdog、cron 和 atd。但部分使用者和开源软件开发者不满意 systemd 的推广模式，并推出了分支 Uselessd 以替代 systemd。

部分开源项目如 Twisted 自带守护程序如 twistd，现在也可以直接使用脚本将 systemd 作为守护程序。

综上所述，要将 Python 脚本或者其他程序设计成守护程序，在不同发行版中可能存在差异。以下是各个发行版的 init 替代软件情况（按字母排序）：

- BootScripts，用于 GoboLinux。

- busybox-init，适用于嵌入式操作系统，OpenWRT 采用，直至被 procd 替代。
- DEMONS，KahelOS 引入的改进版 init 启动流程，守护程序仅在 DE（Desktop Environment）运行后再启动。
- eINIT，init 的完整替代软件，可以异步方式启动进程，也可以在没有 shell 脚本的情况下启动。
- Initng，init 的完整替代软件，可以异步启动进程。
- launchd，Mac OS X V10.4 后开始使用的 init 替代软件。
- Mudur，使用 Python 编写的 init 替代软件，可以异步启动进程，主要用于 Pardas Linux 发行版。
- runit，init 的跨平台替代软件，支持并行启动服务。
- s6，另外一个跨平台 init 替代软件，类似于 runit。
- 服务管理工具，Solaris 10 之后重新设计的 init 替代软件。
- systemd，init 的完整替代软件，可以并行启动服务和其他功能，其已在许多发行版中使用。
- Upstart，Ubuntu 提供的 init 替代软件，可以异步启动进程。

普通的 Linux 软件要设计成系统服务，需要使用 os.fork()，这是一个较为复杂的过程。PEP3143 是目前 Python 的推荐实现，同时在 PEP3143 中也提到了其他实现，如：

- Python Cookbook #66012 和#278731；
- zdaemon；
- dba-daemon；
- clapper-daemon；
- seutter-daemon；
- burr-daemon；
- dagitses-daemon；
- Twisted-daemon。

以上各类守护程序都参考了 *UNIX Network Programming*（作者：W. Richard Stevens）中的实现，而 Twisted 的 twistd 守护程序与其他 Linux 守护程序实现有很大区别。现在以 daemon 库为例实现守护程序。

安装 python-daemon：

```
pip install python-daemon
```

简单的 daemon 代码如下：

```
import daemon

from spam import do_main_program
```

```
with daemon.DaemonContext():
    do_main_program()
```

更加完整的 daemon 代码如下：

```
import os
import grp
import signal
import daemon
import lockfile

from spam import (
    initial_program_setup,\
    do_main_program,\
    program_cleanup,\
    reload_program_config,
    )

context = daemon.DaemonContext(
    working_directory='/var/lib/foo',\
    umask=0o002,\
    pidfile=lockfile.FileLock('/var/run/spam.pid'),
    )

context.signal_map = {
    signal.SIGTERM: program_cleanup,\
    signal.SIGHUP: 'terminate',\
    signal.SIGUSR1: reload_program_config,
    }

mail_gid = grp.getgrnam('mail').gr_gid
context.gid = mail_gid

important_file = open('spam.data', 'w')
interesting_file = open('eggs.data', 'w')
context.files_preserve = [important_file, interesting_file]

initial_program_setup()

with context:
    do_main_program()
```

7.4.11　Linux 定时任务

前面提到 Windows 中采用 schtasks 作为简单定时任务调度。在 Linux 运维中常见的定时任务调度器是 crontab，其守护程序 crond 是系统服务，但是 crontab 任务不能算是系统服务。因

为这些任务并不总是运行的，即并不常驻内存。凡是具备可执行权限的二进制文件、shell 脚本、Perl 脚本、Python 脚本、Lua 等都可以是 crontab job。crontab 必须采用绝对路径调用任务脚本或程序。在 9.10.1.2 节中，会介绍一个 crontab 管理的 Python 库：Plan。该软件包可以大大简化 crontab 的设计。

7.5 移动 APP 开发

在移动互联网时代，移动 APP 开发是重点，而开发 Windows/Android/iOS 兼容 APP 是选择重点。至少从目前的市场占有率来看，需要保证 Android/iOS 两大移动平台的覆盖率。在移动 APP 中采用 HTML+JavaScript 组合的 Web GUI 开发应用，开发周期短，迭代速度快，UI 部分可以持续交付；缺点是响应速度慢，毕竟要运行本地 Web 或者访问远程 Web 资源。此外，其在访问本地资源方面有所限制。原生移动 APP 开发速度较慢，软件迭代和更新与交付必须提交给应用市场审核，比较麻烦；但是其响应速度快，可以访问所有传感器资源。所以现在 APP 开发出现了混合型开发的趋势，取两者之长。

表 7-2 收集、对比了移动 APP 开发中可用的部分开发手段。开发者可以根据需求选择。

表 7-2 移动 APP 开发技术对比表

名　称	语　言	Android	iOS	WP	其他平台	备　注
PhoneGAP	HTML/JS/CSS	是	是	是	Amazon Fire，Tizen/BB10	传感器接口和本地资源
SenchaTouch	HTML/JS/CSS	是	是	是	Tizen	对 iOS 平台兼容性高
Titanium	HTML/JS/CSS/Python/Ruby/PHP	是	是	是		支持云端和本地存储
Intel XDK	HTML+云服务	是	是			兼容 PhoneGAP，访问本地资源
Moto RhoMobile	HTML+Ruby	是	是	是	BB/WM/WP	产生原生 APP，系统底层 API，流畅度很高
Xamarin	C#	是	是	是		
Corona	Lua	是	是		Kindle Fire/Nook	传感器，硬件加速，2D 引擎
LiveCode	拖曳式 IDE	是	是	是	Linux	自然脚本语言，部署工具
Unity	JS/C#/Boo	是	是	是	BB/Xbox/Wii	2D/3D 游戏
MoSync	C/C++/JS/HTML	是	是	是	WM/Java Mobile	本地设备 API 和 UI
Kivy	Python/Cython	是	是	是	Windows/Linux/Mac OS X	多点触摸，GPU 加速

7.5.1　响应式网页

响应式网页，即 Reponsive HTML。作为一种 B/S GUI 技术，它利用 HTML5 和 JavaScript 检测用户设备屏幕和姿态，可以根据屏幕大小、屏幕的横屏和竖起的动作进行切换，甚至可以将菜单进行折叠。这种技术使得一种 Web GUI 设计可以同时支持高清电视、计算机显示器、平板电脑、手机等不同分辨率的设备。

响应式 HTML5 设计是新一代 Web 的标准配置。需要注意的是许多行业的 IT 设备中还在大量使用 Windows XP/Windows 2000。这些行业机构的许多设备，例如医疗设备都是多年前采购的固定资产，价值不菲，更换不易。总之，不是想升级就可以升级的。所以，需要针对这些行业和设备提供专门的 IE 补丁。

7.5.2　PhoneGAP 应用开发

PhoneGAP 开发应该归类于轻 APP 类别的开发方式，其主要的设计迭代发生在服务器 UI 前端。PhoneGAP 是非常流行的一种开发模式。它将浏览器封装在 APP 内，通过服务器端的 HTML5 网页实现华丽流畅的 UI。同时，它还集成了部分本地传感器的资源。不过 PhoneGAP 开发和 Python 没有关系，所以本书不做介绍。

7.5.3　SL4A

SL4A 最初的名称为 Android Scripting Environment（ASE），作者为 Damon Kohler。在 SL4A 之前，Android 的编程语言主要是 Java，而底层 Linux 内核采用 C/C++。SL4A 将各类脚本语言带入了 Android 的应用层开发。

7.5.3.1　SL4A 架构

在 Android 中，应用程序不能够直接访问 Linux 操作系统，因为这在安全性上存在很大漏洞。无论何种脚本语言，它都必须调用操作系统 API 实现底层操作，即访问 Android Dalvik Java VM 提供的 API facade。在 SL4A 设计中，采用了软件设计中的外观模式以及 RPC 通信。图 7-9 揭示了 SL4A 的系统架构和用户脚本运行的流程。

facade，其英文原意如下：外表，外立面，正面，立面，它为建筑学术语。在软件模式中它被译为外观模式：其为子系统中的各类、结构与方法提供一个简明一致的界面，隐藏子系统的复杂性，使子系统更加容易使用。理解其结构可以更好地帮助我们编程，并定位错误。

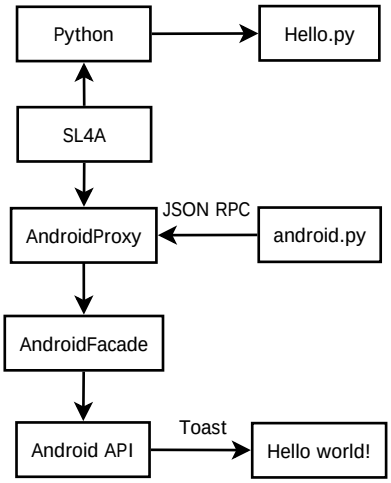


图 7-9　Android SL4A 架构图

SL4A 是一个中间层，位于 Dalvik VM 和应用程序（或脚本）之间，架构类似于分布式计算环境。SL4A 由脚本翻译引擎以及 JSON RPC 服务器构成。SL4A 应用由 Python（或其他语言）用户脚本、第三方 Python 模块以及操作系统访问 API 模块 android.py 组成。其中 android.py 就是 JSON RPC 服务器的编程接口。该文件中定义了一组 proxy 代理函数用于 API 通信，需要导入 android 模块（import android）。API facade 将 Andriod 系统 API 的子集通过 JSON RPC 方式暴露出来，因此这部分权限是系统可控的。

用户脚本通过 SL4A 脚本解释器解释运行。Android UI 中的 toast 是最简单的系统 API，用户脚本如果要调用 toast API，都是通过脚本解释器中的 RPC 调用 API facade 间接地实现系统调用。但是整个流程对开发者都是透明的，从开发者的角度来看，系统是“直接”执行了自己的脚本，弹出 toast 提示。

SL4A 和底层操作系统通信采用 RPC 机制和 JSON 格式；而 RPC 常常用于分布式系统中客户机和服务器之间信息的传递。在 SL4A 架构中，服务器就是 Android OS，SL4A 脚本就是客户端。在 SL4A/Android 中添加的一层 facade，其主要目的是避免有害的脚本危害 Android 系统。

Android API facade 提供了许多 API 接口，请参看表 7-3 了解详情。

表 7-3　SL4A Android API facade

名　　称	说　　明
ActivityResultFacade	设置 Activity 的返回值
AndroidFacade	Android 通用功能
ApplicationManagerFacade	获取已经安装的应用信息

续表

名　　称	说　　明
BatteryManagerFacade	电池管理 API
BluetoothFacade	访问蓝牙功能
CameraFacade	摄像头有关功能
CommonIntentsFacade	Android intent 相关
ContactsFacade	提供联系人访问功能
EventFacade	提供的 API 可以通过 RPC 读取事件队列，并作为纯 Java 函数写入事件队列
EyesFreeFacade	提供 API3 或更低版本的文本语音合成（TTS）
LocationFacade	提供位置管理器相关功能
MediaPlayerFacade	提供基础媒体播放器功能
MediaRecorderFacade	媒体记录、录音、摄像等
PhoneFacade	提供电信业务管理器功能
PreferencesFacade	支持访问偏好设置接口
SensorManagerFacade	提供传感器管理器相关功能
SettingsFacade	提供电话设置功能
SignalStrengthFacade	提供信号强度功能
SmsFacade	提供 SMS 短消息功能
SpeechRecognitionFacade	提供 Android 语音识别功能的相关 RCP 实现
TextToSpeechFacade	提供 API4 或更新版本的 TTS 服务，与 EyesFreeFacade 对应
ToneGeneratorFacade	产生 DTMF
UiFacade	创建并处理获取对话框信息
WakeLockFacade	提供电源管理器的某些功能（唤醒锁定）
WebCamFacade	从前置摄像头获取视频
WifiFacade	管理 Wi-Fi 射频功能

从表 7-3 中，我们可以寻找到最常用的 API。现在的物联网 APP 开发，往往比较关注信号强度、位置管理器、传感器、Wi-Fi/蓝牙和蜂窝数据链接。笔者比较关心 USB Host API facade。实际上，已经有开发者提供了 USBHostSerialFacade。以下是几种可选的方案。

- USB HID/MSD，Android/Linux 已经内置，QPython 应用可以通过 os/sys 标准库获取。
- USB CDC/其他设备类别，采用 USBHostSerialFacade。
- 作为 USBHostSerialFacade 方法的替代方案 A，Android USB host API 可以单独（使用 Java 语言）设计一个服务，并以 TCP 套接字形式提供给其他应用，QPython 应用可以通过套接字标准库配合 RPC 服务获取。这个设计可以参考 pyserial 的设计。

- 作为 USBHostSerialFacade 方法的替代方案 B，采用 Pyjnius 来直接访问 Java API。

7.5.3.2 支持语言

除了 Python，SL4A 还支持许多其他脚本语言，这里做些简单介绍。

BeanShell 是一种 Java 的解释语言。虽然 Android 本身就是用 Java 开发的，但是 BeanShell 解释器提供了一种交互工具来编写和测试代码，而无须像传统 Java 开发那样经历编译、部署、测试的过程。

```
source("/sdcard/com.googlecode.bshforandroid/extras/bsh/android.bsh");
droid = Android();
droid.call("makeToast", "Hello, Android!");
```

Lua，作为一种可嵌入的脚本语言，非常适合 SL4A 的概念。

```
require "android"
name = android.getInput("Hello!", "What is your name ")
android.printDict(name) -- A convenience method for inspecting dicts (tables).
android.makeToast("Hello, " .. name.result)
```

Perl 是 SL4A 所支持的最古老语言之一。

```
use Android;
my $a = Android->new();
$a->makeToast("Hello, Android!");
```

PHP，毫无疑问是最成功的 Web 编程语言之一。

```
<?php
require_once("Android.php");
$droid = new Android();
$name = $droid->getInput("Hi!", "What is your name ");
$droid->makeToast('Hello, ' . $name['result']);
?>
```

Rhino，使得用户可以编写独立的 JavaScript 代码。如果用户需要通过 HTML+JavaScript 来定制 UI，那么可以利用 Rhino 解释器来测试。

```
load("/sdcard/sl4a/extras/rhino/android.js");
var droid = new Android();
droid.makeToast("Hello, Android!");
```

JRuby，在 SL4A 中还不算成熟。

```
require "android"
droid = Android.new
droid.makeToast "Hello, Android!"
```

在 SL4A 中使用 shell 脚本，用户可以使用常见的 Linux 命令：cp/ls/kdir/mv 等。

SL4A 中的 Python 编程部分是我们所关心的。下面是一个最简单的 Python 例子。

```
import android
droid = android.Android()
name = droid.getInput("Hello!", "What is your name ")
print name
droid.makeToast("Hello, %s" % name.result)
Result(id=1, result=None, error=None)
```

国内读者无法从 Google 官网下载 SL4A；不过，GitHub 上有热心的开发者提供了 apk，网址如下：https://github.com/kuri65536/sl4a/releases

SL4A，默认只有 shell 脚本解释器，若需要其他解释器则还是必须单独下载对应的解释器。读者可以根据源码自行编译下载到自己的手机中。

7.5.4　QPython 开发

QPython 可以理解为 SL4A 的 Python 增强定制版。其 APP 的主界面如图 7-10 所示。根据官网信息，QPython 开发者 River 有中文微博： http://weibo.com/qpython。

图 7-10　QPython APP 主界面

QPython 内含 SL4A（Python）引擎、Python 的 REPL 终端（Console）以及 SL4A、Kivy、和 Django 扩展库。为了让开发者可以随时随地开发 Python，QPython 在源码编辑上下了很多功夫。其不仅提供了 REPL 终端，还提供了手机端的编辑器、二维码传输、FTP 上传。其还以演

示代码（qedit4web.py）的形式提供了基于 QPython 的 Web 编辑器，这样用户就可以在任意桌面或移动浏览器中编辑代码，甚至在浏览器中启动脚本，让该脚本在 Android 手机中运行。可惜这个 Web Editor 不能够作为系统服务存在。

QPython 的 SL4A 仅支持 Python 2.7.2，此外还有支持 Python 3 解释器的 QPython 3。

由于 QPython 包含了多个开源项目，如 SL4A/Kivy，但实际能够使用的 SL4A API facade 文档在 QPython 官网上没有，因此需要参考这个网址：http://kylelk.github.io/html-examples/androidhelper.html。

我们可以在 Console 中先测试一下 Android 设备。

```
>>> import androidhelper
>>> droid = android.helper.Android()
>>> droid.vibrate(300) # vibrate for 300ms
Result(id=6, result=None, error=None)
>>> droid.checkBluetoothState()
Result(id=7, result=True, error=None)
>>> droid.checkWiFiState()
Result(id=8, result=True, error=None)
>>> droid.phoneCall('13801235678')
Java.lang.NullPointerException
Result(id=9, result=None, error=u'Java.lang.NullPointerException')
>>> droid.getCellLocation()
Result(id=10, result=None, error=None)
>>> for i in dir(droid):
...    print i
```

很遗憾，虽然文档上列举了许多功能，但是实测却返回 None，或者直接抛出空指针错误。有可能笔者使用的 QPython 构建时的版本和权限有关。感觉上，QPython 项目比较集中测试了 Web GUI 方面，但是和手机相关的硬件和服务没有得到足够的重视。

QPyPI

QPython 和桌面版 Python 类似，它也有自己的组件管理工具：QPyPI。它的网址是 http://qpypi.qpython.org/。除了 QPyPI，QPython 应用也可以从 Python 官方 PyPI 及镜像站点下载扩展模块。

在 QPython QPyPI 初始页面中，最流行的四个下载项目分别是 multiprocessing、NumPy、plyer、sqlite3。要查看其他项目，直接单击“search”按钮，就可以获取 QPython 库列表。

使用者可以在 QPython 的 QPyPI 管理器中，或者在终端运行 pip 脚本来下载并安装各种组件。纯 Python 库基本上都可以使用。但在实际使用过程中，笔者发现 QPython 在处理一些压缩安装包中存在问题，比如：

```
pip install twisted
Downloading Twisted-16.2.0.tar.bz2

CompressionError: bz2 module is not available
```

这说明，在 Android 中缺乏 bzip 支持，QPython 需要重新编译增加 bzip 支持。同样的命令，在另外一台 Android 手机上运行正常。提示读者特别注意这一点。

Django 在 QPython 中主要作为 Web GUI 的本地服务器来使用。通过运行本地（手机）内的 Django（或使用更加轻量级的 Flask/webpy/bottle）服务器，将浏览器封装为前端 GUI 来构建本地 APP。当然，Web GUI 方式更加适合一些和网络操作有关的应用：比如文件传输等方面的 APP 和即时消息等。由于 SL4A 是作为 Android 应用而存在的，因此此类 Python 服务器和 Linux 服务器中的系统服务是有区别的。但 SL4A 依然可以完成部分系统服务的工作。

此外，QPython 的所有程序一般作为 QPython 应用的一部分，必须安装 QPython 后，在 QPython 宿主 APK 中运行用户脚本或项目。其比较适用于那些中小型的开源项目。

QPython 应用也可以打包成 APK 成为独立运行的应用，但是其比较复杂。在 PyCon 2015 中国年会上，为了简化整个流程，QPython 发布的 APKbuilder 云端服务提供了一条龙打包服务：提交图标、名称、应用 ID、main.py 文件和邮箱，QPython 可以将打包后的 APK 发回用户注册邮箱。

顺便提一句，百度贴吧有专门的 QPython 吧，不过不活跃。笔者观察下来发现，QPython 开发者远少于 Java 开发者。

笔者使用之后发现，QPython 中的 Kivy 并非完整版本。开发手机 APP 的读者需要安装配置 Kivy 开发环境、编写 Python 代码、调试，并打包成 Android APK 安装包。接下来，专门针对 Kivy 的 NUI 开发进行介绍。

7.5.5　Kivy

Kivy 是一个开源工具包，能够让相同源代码创建的程序跨平台运行。它主要关注创新型用户界面开发，如多点触摸应用程序。Kivy 还提供一个多点触摸鼠标模拟器。在 Android 的 SL4A 和 QPython 中，Kivy 的例程被包含在内。同时，Kivy 是一个迎合了桌面计算和移动计算融合趋势的平台。图 7-11 是 Kivy 徽标和它目前支持的操作系统平台徽标。

注意　QPython 和 Kivy 是两个独立的开源工程。QPython 在 SL4A（Python）基础上包含了 Kivy 的一些设计和例子，但没有完整包含 Kivy 的所有类。

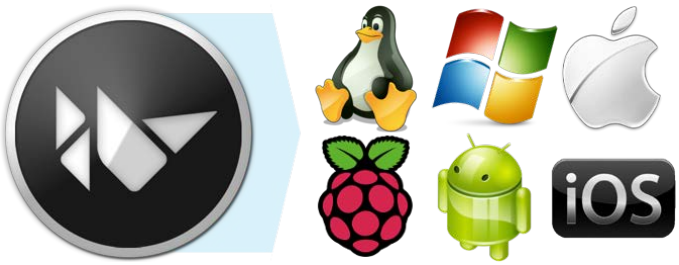


图 7-11　Kivy 徽标与其支持的操作系统平台徽标

7.5.5.1　主要特点

跨平台

Kivy 可运行于 Linux、Windows、OS X、Android 和 iOS。同样的代码可以运行于其所支持的所有平台。其原生支持大多数输入方式、协议和设备，包括多点触摸鼠标模拟器。

商业友好

Kivy 可 100%免费使用，适用于 MIT 和 LGPL3 许可证。其工具箱设计专业，可以用于商业产品。其 UI 框架很稳定，文档齐全，加上编程指南，可以轻松开发。

GPU 加速

图形引擎基于 OpenGL ES2，采用最新的快速图形流水线。工具箱附带二十多个高度扩展的 Widget。多数组件使用 Cython C 语言编写，并经过回归测试。

注意　Kivy 基于 Cython（C extensions for Python）构建，而非 CPython。所以，在各个操作系统中的安装需要花点儿心思。

计算机系统的人机界面经历了若干代的进化。

（1）CLI：Command Line Interface，命令行界面；

（2）GUI：Graphic User Interface，图形人机界面；

（3）NUI：Natural User Interface，自然用户界面。

NUI 是以多点触摸、语音识别和交互为特征的符合人类使用习惯的输入/输出用户界面。Kivy 是 NUI 的开发框架。

7.5.5.2　APP 生命周期

从图 7-12 的 Kivy 应用程序生命周期图中可以看出来，虽然 Kivy 是跨平台的 NUI 框架，但是 Kivy 应用程序与 Android APP 的生命周期非常类似，说明其受到 Android 的影响很大。

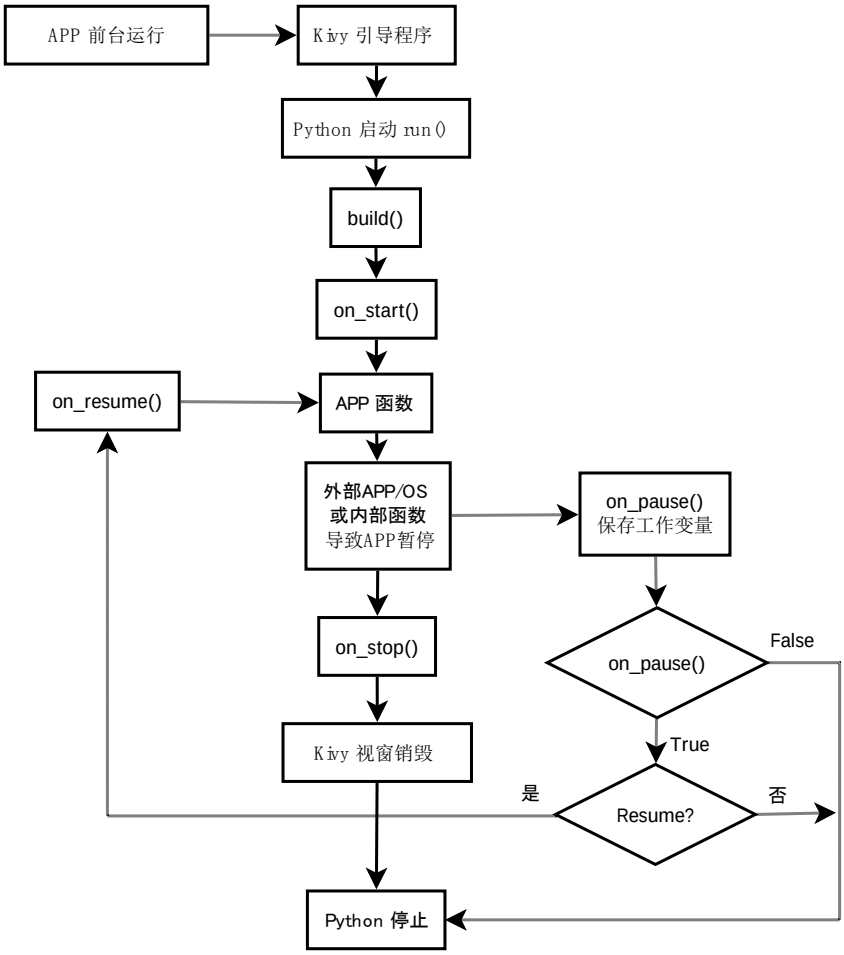


图 7-12　Kivy 应用程序生命周期图

7.5.5.3　Kivy UI 脚本运行流程

下面我们以Android QPython内置Kivy UI来做一些设计。首先在编辑器中写下以下演示代码。

```
#qpy:kivy
'''
Kivy_demo.py
This is a demo for Kivy
'''
import kivy
kivy.require('1.0.6')

from kivy.app import App
from kivy.uix.button import Button
```

```
class TestApp(App):
    def build(self):
        return Button(text='Hello World')

TestApp().run()
```

运行 Android 手机中的 QPython。选择主页面中的“关于|FTP 服务器”菜单项，运行 FTP 服务器。注意 FTP 端口、用户名和密钥。利用 FTP 上传工具将 Kivy_demo.py 上传到手机 FTP 服务器的 script 路径中。返回主界面后，选择程序，并选中 Kivy_demo.py，之后运行。手机会出现一个占满全部屏幕（不包含通知条）的 HelloWorld 例程。

7.5.5.4 kv 资源文件

在 Adnroid UI 开发中可以使用 XML 文件来定义 UI 资源，类似的概念在 wxPython 中也有。Kivy 也使用自定义的 kv 文件来定义 UI 资源。

test.kv：

```
#:kivy 1.0
Button:
text:'Hello from test.kv'
```

main.py:

```
import kivy
kivy.require('1.0.7')
from kivy.app import Appclass

TestApp(App):
    pass

if__name__=='__main__':
    TestApp().run()
```

7.5.5.5 传感器等本地资源

HTML5 APP 和原生 APP 的主要分野之一就是本地资源的访问能力。这从 API 文档中的 Kivy 配置项就可以看出端倪。

```
kivy.kivy_options = {'camera': ('opencv', 'gi', 'pygst', 'videocapture', 'avfoundati
on', 'android'), 'window': ('egl_rpi', 'sdl2', 'pygame', 'sdl', 'x11'), 'spelling':
('enchant', 'osxappkit'), 'audio': ('gstplayer', 'pygame', 'gi', 'pygst', 'ffpyplaye
r', 'sdl2', 'avplayer'), 'text': ('pil', 'sdl2', 'pygame', 'sdlttf'), 'clipboard':
('android', 'winctypes', 'xsel', 'xclip', 'dbusklipper', 'nspaste', 'sdl2', 'pygame',
 'dummy', 'gtk3'), 'video': ('gstplayer', 'ffmpeg', 'ffpyplayer', 'gi', 'pygst', 'py
glet', 'null'), 'image': ('tex', 'imageio', 'dds', 'gif', 'sdl2', 'pygame', 'pil', '
ffpy')}
```

Kivy 对于 Android 摄像头、音频、文字（字体）、剪贴板、视频、图像提供了一定程度的支持；而其对于传感器和移动通信缺乏足够的支持，需要其他第三方类如 plyer 补足。

Python 程序可以通过 Pyjnius 来访问 Android 中的 Java 类。还可以通过 plyer 来扩展 Kivy 手机 APP。plyer 是一个与平台无关的 Python 封装器，封装了一些平台独立的 API，可以读取传感器、发送电邮、文本转语音服务、显示通知等功能的库。

```
from plyer.vibrator import vibrator
vibrate(10) # for 10 seconds
```

根据表 7-4 所列开发状态显示，plyer 支持的最完整的系统为 Android。plyer 会根据平台的不同而调用不同的外部库：

- python-on-android，调用 Pyjnius；
- kivy-ios，调用 pyobjus；
- 桌面操作系统 Windows、Linux、OS X，会使用常见的标准库和第三方库。

表 7-4　plyer 开发状态表

平台	Android	iOS	Windows	OS X	Linux
加速度计	是	是		是	是
电话业务	是	是			
摄像头（拍照）	是	是			
GPS	是	是			
通知	是		是	是	是
TTS	是	是	是	是	是
电邮	是	是	是	是	是
振荡器	是	是			
SMS（发送）	是				
指南针	是	是			
UID	是	是	是	是	是
陀螺仪	是	是			
电池	是	是	是	是	是
本地文件	是		是	是	是
方向传感	是				
录音	是				
闪光灯	是	是			

7.5.5.6 Pyjnius

目前，USB 设备尚无法得到 PhoneGap 和 Kivy/plyer 的支持。所以，还需要 Pyjnius 工具来实现 Python 在 Android 下访问 USB 设备。但是这需要开发者对于 Android Java 类非常熟悉，而且对 USB 主机开发也非常熟悉。这对某些嵌入式应用还是很有意义的。

以下是通过 Pyjnius 来记录声音的例子。

```
from jnius import autoclass
from time import sleep

# get the needed Java classes
MediaRecorder = autoclass('android.media.MediaRecorder')
AudioSource = autoclass('android.media.MediaRecorder$AudioSource')
OutputFormat = autoclass('android.media.MediaRecorder$OutputFormat')
AudioEncoder = autoclass('android.media.MediaRecorder$AudioEncoder')

# create out recorder
mRecorder = MediaRecorder()
mRecorder.setAudioSource(AudioSource.MIC)
mRecorder.setOutputFormat(OutputFormat.THREE_GPP)
mRecorder.setOutputFile('/sdcard/testrecorder.3gp')
mRecorder.setAudioEncoder(AudioEncoder.AMR_NB)
mRecorder.prepare()

# record 5 seconds
mRecorder.start()
sleep(5)
mRecorder.stop()
mRecorder.release()
```

由此可见，在一些特定应用中，原生语言的地位依然还是最重要的。即使是采用 Python 这种易学易用的语言来编写 APP，有时候也需要了解更多原生语言的细节。

7.5.5.7 Kivy 应用打包

Kivy 是跨平台的 NUI，所以针对不同平台的打包形式是不同的。

- Windows：PyInstaller；
- Android：Buildozer/Python-for-android；
- OS X：Buildozer/PyInstaller/Homebrew；
- IOS：Xcode。

7.5.5.8 Kivy 开发环境配置

Kivy 的移动应用打包其实是挺复杂的，我们主要说一说 Android 应用打包。在此需要用到 Python-for-android 和 Buildozer。

- Python-for-android：专为在 Android 设备中发布 Python 应用而设计的开源工程，包括所需的模块，创建包括 Python、库和用户应用在内的 APK 打包应用。
- Buildozer：构建 Android 应用的自动化工具。它会自动下载并为 Python-for-android 配置好所有的依赖项，包括 Android SDK/NDK。构建好 APK 后会自动推送到设备中去。

以上两个都是基于 Linux 的工具。为了进一步推广和简化开发环境的配置，Kivy 特别提供了一个 VirtualBox 的虚拟机文件，文件大小大约为 2GB。这样一来，开发者可以不必自行搭建开发环境，可以直接在虚拟机中开发，实现原生 Android APK 的打包。该虚拟机文件中仅仅包括了 Python-for-android 项目。所以，开发者除了使用该虚拟机文件外，还可以使用 Buildozer 工具来自动化整个过程。对于新手开发者，推荐使用 Buildozer，其最容易构建完整 APK。不喜欢 Java 编译的开发者请使用 Kivy Launcher。该工具可以运行 Kivy 程序而无须编译代码。以上仅仅是打包流程。Kivy 应用发布到 Android 市场前，需要额外的签名步骤。

7.5.6　其他开发方式

在移动 APP 领域，还有许多其他敏捷开发方式。不过，其大多数都是基于 HTML/CSS/JavaScript 或 Java 开发的。常见的有：

- 百度轻 APP；
- 基于 HTML5 的轻 APP 开发——APICloud、轻 APP、liveApp、云起等。

这些 APP 开发可以直接在 Web 上开发，还可以托管在这个服务商云端，然后利用 PhoneGap 等进行封装。Google 也有一些 APP 开发的云端服务，可以减少开发原生 APP 的工作量。

7.6　本章小结

本章针对桌面和移动端应用程序的 Python 开发、打包、安装技术做了一些梳理。Python 桌面开发的特点是跨平台，而移动端开发并不占优势。更多应用还是使用原生语言进行开发，或者采用 HTML5 开发轻 APP。

从工程实践的角度来看，选择最稳妥的主流技术方案是最现实的选择。Python 开发应用虽然有许多方法和手段，但开源软件往往依赖于开发者自己的动手能力和钻研精神。

综上所述，桌面应用开发推荐 PyQt/PySide/wxPython 作为优先考虑方案。因为这些技术来源丰富，也有足够的源码可以借鉴、参考。此外，需要关注以 Web GUI 为基础的方案，因为这是一种很重要的技术趋势。

而对于移动应用工程，应该优先考虑 Java 等原生语言和 PhoneGAP 等 Web GUI 方案。关于 QPython/Kivy 技术方案，开发者除了掌握 Python 语言外，还需要了解底层 Java 开发和 Android OS API 细节。这对于开发者的要求反而更高了。开发者刻苦钻研、掌握这些技术之后，才能够将其真正地应用到实际工程开发中。

第 8 章

Python 开发辅助支持

在物联网开发链条中，开发者要进行数据分析、算法设计、原型验证、自动化测试、辅助工具设计。每个环节的工作效率对整体的开发进度和质量都非常重要。选用一个好的辅助开发工具（集）是非常必要的。Python 容易上手，应用广泛，拥有众多功能强大的第三方库，开发平台轻量化，非常适用于嵌入式和服务器端开发中的辅助开发。

所谓开发支持，指的是开发过程需要的各类生产辅助工具。Python 的胶水语言特性使得它可以在物联网诸多开发环节和层面上构建不同的工具。下面举几个例子。

1．原型验证

原型验证主要是使用通用软件和硬件来模拟要开发的系统。由于 Python 实现起来比其他语言快，因此一些高层建模和算法可以利用 Python 的科学计算包来模拟。算法验证通过后，再逐步优化，实施性能加速，换用或者混用原生语言进行开发。

2．代码生成

Python 支持面向对象编程，且具有很强的文档操纵能力；Python 在导入 XML/CSV 或其他模型文档后，可以生成代码对象，并产生目标源代码。目标语言可以是 C/C++，也可以是 Java 或者其他 XML 格式文件。

除了直接参与代码与工程外，Python 还可以用于系统质量体系。软件工程和质量管理中最常见的是软件测试、文档管理和配置管理。

3．自动测试

软件工程中提升软件质量的重要手段是软件测试。Python 包含许多测试框架，其不仅仅可以用于测试 Python 的代码质量，还可以用于测试 Java 的代码质量，并进行 IC 的单元测试。由于 Python 的敏捷开发能力和越发复杂的测试要求，因此嵌入式系统也开始利用 Python 做自动测试。

4．文档生成

Python 内置 docstring，支持从代码中提取代码，这一点和 Javadoc 一样。有大量的 Python 包可以将各种文档转换成出版级别的图形和文档。此外，Python 也支持文档和代码中常见的国

际化，并有本地化支持。

5．配置管理

软件工程中配置管理是日常工作之一，Python 脚本可以实现程序化访问 Git/Subversion/CVS，采用自动策略进行软件配置管理，这在持续交付和自动化运维方面需求非常旺盛。同时还可以帮助软件质量工程师进行自动化检查修改。

6．数据获取与处理

在许多物联网应用中，数据必须先进行清洗，整合，转换成后端系统可以利用的格式。无论是网络爬虫还是大数据分析，数据集成都需要进行这一步的处理。

7．虚拟仪器

嵌入式开发的范围其实很宽泛。在开发中，可能需要各种各样的开发工具。除了 C 编译器、仿真器、万用表外，最好还能够配备示波器、逻辑分析仪、频谱分析仪、音频信号发生器等，某些时候甚至需要采购一些专用的定制设备。在工程实践中，资金预算总是不够。Python 配合一些开源硬件可以组成一些虽然简陋但却很实用的虚拟仪器。此外，Python 也可以直接访问专业仪器仪表。

物联网不断发展，相关工具也层出不穷。笔者会专门收集这方面的信息，通过专门的网页和博客与读者分享这些实用工具资源，并在实际工程设计中实现加速开发。

8.1 物联网开发需要不断优化

在物联网产品或服务框架的开发过程中，许多工作流程都可以利用软件进行自动化。这要求我们经常评估手头工作，观察哪些工作或流程可以被工具化。俗话说“磨刀不误砍柴工”，利用 Python 开发这些工具，其长期收益往往会高于研发代价。开发并充分利用这些工具，可以加速开发的迭代速度。

这类优化可以是代码层面的优化，如使用更通用和抽象的数据结构和编程技巧来减少重复代码；也可以是工具层面的优化，如使用代码生成器自动生成代码，使用自动测试工具简化和加快测试；还可以是流程级别的优化，比如提供工具来实现设计流程扁平化，让客户和合作伙伴可以部分参与设计流程。

8.2 专属小工具

所谓专属小工具指的是一些针对特定目的的工具软件。

8.2.1　单位转化器

图 8-1 是 NXP LPC81X 数据手册中的 DIP8 封装说明图。NXP 的数据手册算是完整的，其机械尺寸有英制和公制两套参数。硬件工程师的日常任务除了绘制 PCB 外还要制作元件库。数据手册现在往往使用公制单位，而大多数工程师和 CAD 软件都偏向英制单位。制作元件库时常常涉及公制毫米（mm）和英制密尔（mil）的数据转换。两种单位的转换很简单：

```
1 mm = 39.37 mil
100 mil = 2.54 mm
```

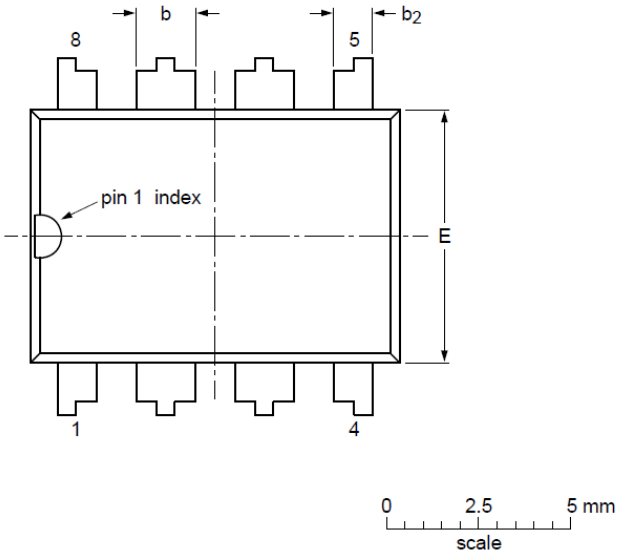


图 8-1　DIP8 封装说明图

在大多数情况下，工程师会记住两者之比大约是 1：40。该比例适用于精度要求不高的例子，常见的双列直插如 DIP8 就是一例。但现在许多 PCB 设计中的元件封装，如 CSP/WLCSP 等精度要求越来越高，需要计算累计误差。在设计元件库时反复进行英制与公制的转换计算让人厌倦，所以笔者在做 PCB 元件库时，随手编写了一个 Python 程序 mmmil.py。

```
#!/usr/bin/env python
import string
import sys

print "Converter between mm and mil"
print "Please enter [digits][space][unit], like 100 mil or 10 mm"
rate = 0.0254
```

```
def mil2mm(val):
    return val*rate

def mm2mil(val):
    return val/rate

while(True):
    choice = raw_input()
    try:
        (origin, unit) = choice.split(' ')
    except ValueError,e:
        print "Invalid format, please enter space between digit and unit"

    if origin.upper() == "Q":
        exit(0)

    if unit.upper() == "MIL":
        print "%f mm"%(mil2mm(float(origin)))
    elif unit.upper() == "MM":
        print "%f mil"%(mm2mil(float(origin)))
    elif unit.upper() == "CM":
        print "%f mil"%(mm2mil(float(origin)*10.0))
    else:
        print "Invalid unit"
```

这种工具性代码在日常开发中非常多。通常我们会在网络上寻找软件，但是许多细小的特定需求往往没有对应的软件，而利用 Python 可以很快地进行开发。

8.2.2 内码转换器

许多时候，我们设计的设备或电子产品面对的是全球市场，需要支持多国语言界面。在笔者开发过的液晶电视参考设计中，其 OSD 菜单采用多达 36 国语言。除了泛欧拉丁语系如英语、德语、法语、意大利语、葡萄牙语、西班牙语、荷兰语、瑞典语等外，还有采用西里尔字母（Cyrillic）的斯拉夫语系，如俄语、乌克兰语等，以及波斯语和阿拉伯语，汉语、韩语、日语、越南语等亚洲语言。早期销售的液晶电视没有内置 Android 系统，最多也就是 8051 或者 ARM+Linux 组合的芯片组，资源相对有限，所以需要专门维护点阵字库。这种情况目前在许多终端中依然大量存在。

在计算机系统中，采用 unicode 可以支持全部的语言文字。unicode 字库基本上都是 TTF 字库，文件大小均以 MB 计算。所以，unicode 无法直接在资源受限的嵌入式系统中使用。嵌入式 UI 系统往往基于字符点阵技术，而不是矢量字体。随处可见的 LED 条屏、电视机、显示器、无绳电话和许多工业设备等都是如此。

除了 unicode 字体无法保存外，CKJV（Chinese/Korean/Japanese/Vitnamese，即汉/韩/日/越）

等字体还存在无法在设备中包含全部字符的问题。所以，在设备中往往有自定义的小规模字库，并将字库中的点阵信息复制到小字库中去。此外，波斯语和阿拉伯语的书写方向和变化，决定了其必须采用点阵拼接方式，也需要定制小字库。

最原始的方式是找到对应语言的点阵字库，人工绘制字库，并嵌入代码中。但其效率低，容易出错。相对有效率的做法如下：

（1）通过 unicode 编码字符串提取对应字符字体；

（2）将 TTF 字库转换成对应矩阵的点阵字库；

（3）将点阵字库扫描方式进行转换；

（4）将字库信息合成在小字库中，嵌入代码中。

在实际工程中，由于许多 OSD 器件的点阵要求，导致以上的方法转换出来的字库不可用。这主要体现在：

- 低于 32×32 点阵解析度的汉字扭曲变形，不美观；
- 英文字库往往采用 5×7 点阵，而汉字采用 12×12、16×16、24×24 和 36×36 点阵。

所以，低于 32×32 点阵的字体提取的替代方式如下：

- 通过 unicode 编码字符串提取对应语言内码字符串，比如从 unicode 转换成 GB2312 内码；
- 提取各个语言的老式点阵字符，比如根据 GB2312 内码从 HZK 字库提取点阵字符；
- 将点阵字库进行各种转换，如大小端、纵向扫描、横向扫描、纵横扫描顺序等；
- 将字库信息合成小字库，转换成 C/Java 数组嵌入代码中。

不同语言，包括 CKJV、泛欧语系、斯拉夫语系都有不同的内码，所以在不同内码间及 unicode（UTF8/UCS2 等）之间进行转换也是常见的开发需求。

8.2.3　其他编码转换

除了人机界面中由于各种原因在 unicode 和各国语种系统内码间进行转换外，计算机系统中还经常进行通信协议的转换。GSM 的 PDU 编码就是典型的专用编码格式，此外还有电子邮件的 UUEncode、URL 编码采用的 Base64 等编码。这里贡献一个 GSM 编解码的例子，代码引自 GitHub pducodec 工程：

pducodec_demo.py：

```
import PDUSMScodec

usrText = "allankliu says hello to you."
pduText = "61363BEC5EB3D375D03C9C9F83D06536FB0DA2BF41F977DDD55000"

print "UsrText(%dB):\t%s"%(len(usrText),usrText)
pduEncode =  PDUSMScodec.gsm_encode("allankliu says hello to you.")
```

```
print "PDUEncode(%dB):\t%s"%(len(pduEncode),pduEncode)

print "PduText(%dB):\t%s"%(len(pduText),pduText)
pduDecode = PDUSMScodec.gsm_decode(pduText)
print "PDUDecode(%dB):\t%s"%(len(pduDecode),pduDecode)
```

PDUSMScodec.py：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
gsm = (u"@£$¥èéùìò?\n??\r??Δ_ΦΓΛΩΠΨΣΘΞ\x1b???é !\"#¤%&'()*+,-./0123456789:;<=>"
   u"??ABCDEFGHIJKLMNOPQRSTUVWXYZ???ü`?abcdefghijklmnopqrstuvwxyz???üà")
ext = (u"````````````````````^````````````````{}`````\\````````````[~]`"
   u"|````````````````````````````````````€````````````````````````")

def get_encode(currentByte, index, bitRightCount, position, \
    nextPosition, leftShiftCount, bytesLength, bytes):
    if index < 8:
        byte = currentByte >> bitRightCount
        if nextPosition < bytesLength:
            idx2 = bytes[nextPosition]
            byte = byte | ((idx2) << leftShiftCount)
            byte = byte & 0x000000FF
        else:
            byte = byte & 0x000000FF
        return chr(byte).encode('hex').upper()
    return ''

def getBytes(plaintext):
    if type(plaintext) != str:
        plaintext = str(plaintext)
    bytes = []
    for c in plaintext.decode('utf-8'):
        idx = gsm.find(c)
        if idx != -1:
            bytes.append(idx)
        else:
            idx = ext.find(c)
            if idx != -1:
                bytes.append(27)
                bytes.append(idx)
    return bytes

def gsm_encode(plaintext):
    res = ""
    f = -1
    t = 0
    bytes = getBytes(plaintext)
```

```
    bytesLength = len(bytes)
    for b in bytes:
        f = f+1
        t = (f%8)+1
        res += get_encode(b, t, t-1, f, f+1, 8-t, bytesLength, bytes)

    return res


def chunks(l, n):
    if n < 1:
        n = 1
    return [l[i:i + n] for i in range(0, len(l), n)]

def gsm_decode(codedtext):
    hexparts = chunks(codedtext, 2)
    number   = 0
    bitcount = 0
    output   = ''
    found_external = False
    for byte in hexparts:
        byte = int(byte, 16);
        # add data on to the end
        number = number + (byte << bitcount)
        # increase the counter
        bitcount = bitcount + 1
        # output the first 7 bits
        if number % 128 == 27:
             '''skip'''
             found_external = True
        else:
            if found_external == True:
                character = ext[number % 128]
                found_external = False
            else:
                character = gsm[number % 128]
            output = output + character

        # then throw them away
        number = number >> 7
        # every 7th letter you have an extra one in the buffer
        if bitcount == 7:
            if number % 128 == 27:
                '''skip'''
                found_external = True
            else:
                if found_external == True:
                    character = ext[number % 128]
```

```
                found_external = False
            else:
                character = gsm[number % 128]
            output = output + character

        bitcount = 0
        number = 0
    return output
```

由于这段代码在主机上测试下来没有问题，因此笔者将其转换成 mbed C/C++代码，移植到 M2M 项目的 MCU 固件中。

8.3 原型验证

互联网创业提倡“精益创业”，这里有一个重要的方法，就是 MVP（Minimum Viable Product，即最小可行产品）。其核心思想就是开发产品先做出一个简单原型，然后进行测试并收集用户反馈，之后进行快速迭代，修正产品。

统计数字表明，软件系统中 15%的错误起源于错误的需求。为了提高软件质量，确保软件开发成功，降低软件开发成本，一旦对目标系统提出一组要求之后，就必须严格验证这些需求的正确性。

Python 非常适合快速构建各类原型，尤其是与算法相关的计算，如数字信号处理、加密、认证、科学计算等，利用 Python 开发原型远比 C/Java 效率高。虽然 CPython 的性能不高，与 C/C++/ Golang 相比差 1~2 个数量级，但可以使用其他 Python 实现并通过各种加速方式提升性能，还可以混用其他编程语言来突破性能瓶颈，以逐渐将 Python 系统设计算法落实到生产环境中并进行持续优化。

在物联网开发中，系统从小开始发展，在许多环节一开始都没有实施完毕的情况下，需要依靠一些模拟设备来做测试和开发。举例来说，物联网开发往往基于 M2M+设备云+APP 的模式。M2M 固件开发需要一个小型设备云，而设备云需要一个虚拟 M2M 设备；同样，APP 也需要一个设备云提供 REST API 和加密通道来测试。而 Python 可以用来构建快速原型，在较短时间内构建小型设备云和虚拟设备以满足 M2M/设备云/APP 三个环节的并行开发需求。

以上是许多供应商所谓的设备虚拟化。这还带来一个附加的好处：**分清组织机构内部的责任，减少互相推诿问题**。由于虚拟设备和实际设备的共存，这使得交叉检验成为可能，可以作为联调与定位错误的基础。设备或服务器的软件 bug 可以相对容易地定位在某个环节和工程师之上，这在物联网的数据传输和系统整合中非常有意义。

8.4　代码生成器

所谓代码生成器，就是利用程序和代码模板来自动生成源代码，减少程序员的工作量，这可以在需要频繁修改程序，而人工编写比较容易出错的源码构建环节中大量使用。我们日常使用的 Web 前端框架编译器、GUI 可视化编程，都可以说是代码生成器的例子。比如前面提到的 wxPython/wxGlade 工具就是一个例子。

现在许多嵌入式系统也有自己的用户界面，如 Qt Embedded，通过可视化编程，大大减轻了 UI 的编写。可惜并不是所有 GUI 都可以拥有这种开发环境。尤其在嵌入式领域，更是如此。

解决方式之一就是使用代码生产器：

（1）建模，即编写代码生成器代码；

（2）参数配置，输入所需配置信息，可以采用文本、CSV、XML 或 JSON 定义配置信息；

（3）生产代码，通过代码生成器，输出目标源代码；

（4）编译链接，将源代码编译链接，形成最终目标代码。

一般来说，在以 LCD/LED/OSD 为主的菜单设计中，菜单都是通过某种结构体数组来定义的，并由 UI 主循环对 GUI 菜单进行遍历和处理。我们针对结构体数组编写 Python 脚本产生 C 源码和 H 头文件。此类脚本主要用于产生最初版本的源码和头文件，通常不会在源码和头文件上做升级修改。作为一种替代方法，需要在源文件中自定义一种标记，让脚本可以识别并通过数据库（如 SQLite 或者 XML）进行管理。

在笔者早年于显像管彩电业使用汇编语言开发固件的时代，就已经使用 gawk 工具来进行代码生成了。gawk（GNU awk），是 UNIX 中的文本查找和替换工具。当时的主要开发语言是宏汇编，gawk 可以用来创建一些应用相关的数据结构和定义。但 gawk 的语法比较简单，笔者后来觉得 Perl 的文本操控能力比较强，于是升级到了 Perl。用了一段时间之后，笔者发现自己容易被 Perl 如同咒语一般的语法所迷惑，维护起来很麻烦，就启用了 Python。Python 有着类似于 C 的语法，支持面向流程和面向对象的编写方式，嵌入式固件工程师可以自己维护，并支持自动测试。之后在类似的应用场景中，笔者逐渐增加了 Python 代码生成器的使用范围。

代码生成器的其他使用场景如下。

- 存储器静态划分，包含字段划分、初始化；解决了修改不同存储位置，需要在宏定义多处修改的问题。
- 接口定义：如芯片间、设备间的自定义通信协议。
- 小字库构成，将使用到的中文字库点阵整合，并合并为一个 ROM 数组。

这些代码都从源代码角度，先扫描用户配置信息，然后产生目标源文件。现在看来应该从对象角度来看，因为 Python 的数据类型远比 C/C++要多，所以可以先自定义类，然后提取用户数据进行填充，之后进行对象管理。以 EEPROM 文件系统为例，应该定义 EEPROM 类，实例

化后进行用户数据填充，并生成器件内子地址，继而形成 C 源码所需要的头文件和宏定义。

芯片定义

本节所描述的内容是笔者的一个互联网服务构想：利用 XML 定义芯片元数据，并用于产生文档、封装、模型和代码，就像描述 GUI 的 XML 资源文件一样。

芯片专用的 XML 格式可以描述芯片的各类接口：寄存器、引脚、封装尺寸、仿真模型。配合 CSS/XSLT 样式表，可以很容易导出为各类文档（XLSX、DOCX、XHTML、PDF、SVG 等），并进一步利用 Python 脚本（或其他工具）来产生对应的文档、C/C++编程所需的头文件、函数原型，甚至软件模块。重要的是，这些文档和源码是可以复用的。

现在的 IC 寄存器动辄几十个甚至成百上千个寄存器，引脚也都呈几何级数上升。XML 文档产生后，可以作为其他文档的数据源，这样可以大大节省工程师在文档、软件、硬件和现场支持方面的工作量，并提升文档和软件质量。由于 XML 是可以检索的，因此可以将 IC 相关结果保存在云端，除了检索 IC 本身的各种属性和文档，还可以检索相关开发资源链接、供货情况、替换元件和竞品情况。

XML 定义中需要规划许多属性和元素，以下仅仅是一个草稿。

```
<?xml version="1.0" encoding="utf-8"?>
<ic>
    <name>PCF8563</name>
    <supplier>NXP Semiconductors</supplier>
    <register>
        <reg width="8bit">
            <addr>0x00</addr>
            <regbit bit="0" reset="0"/>
            <regbit bit="1" reset="0"/>
            <regbit bit="2" reset="0"/>
            <regbit bit="3" name="TESTC" reset="1"/>
            <regbit bit="4" reset="0"/>
            <regbit bit="5" name="STOP" reset="1"/>
            <regbit bit="6" reset="0"/>
            <regbit bit="7" name="TEST" reset="1"/>
            <regbit></regbit>
        </reg>
    </register>
</ic>
```

由于大多数数据手册的结构是类似的，因此可以利用网络爬虫或者搜索引擎下载 PDF 文件，并使用 PDFMiner3K 从 PDF 中提取信息，之后将 IC 特性表和订购信息自动输入到数据库，在互联网中共享。

8.5　软件测试

测试是一项很繁重的工作，所以需要一种开发便利的语言做测试工作。Jython 的应用目的之一就是利用 Python 的敏捷开发能力来测试 Java 代码。Python 的快速测试能力已经用在了许多场景，包括汽车电子等重度依赖软件质量和自动测试的场合。所以，利用 Python 对嵌入式设备、服务器和其他软件产品进行测试的需求会被许多其他领域的开发者使用。软件测试有如下分类。

- 单元测试：这是指对软件基本组成单元，即软件设计的最小单位进行正确性检验的测试工作，如函数、过程（function，procedure）或类的方法（method）。
- 集成测试：这是指在单元测试的基础上，将所有模块按照概要设计要求组装成为子系统或系统，验证组装后的功能以及模块间的接口是否正确的测试工作。集成测试也叫组装测试、联合测试、子系统测试或部件测试。
- 系统测试：这是指将已经集成好的软件系统，作为整个计算机系统的一个元素，与计算机硬件、外设、某些支持软件、数据和人员等其他系统元素结合在一起，在实际使用环境下，对计算机系统进行一系列的组装测试和确认测试的工作。
- 覆盖测试：这是指被测试系统的测试覆盖程度，即一项给定测试或一组测试对某个给定系统或构件的所有指定测试用例进行测试所达到的程度。
- 回归测试：回归测试是指修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码产生错误。

不同测试阶段会用到不同的测试方法，如表 8-1 所示。

表 8-1　测试阶段和方法

测试阶段	测试方法	简　　介	评估基准
单元测试	白盒测试	单元内部的数据结构、逻辑控制、异常处理等	逻辑覆盖率
集成测试	灰盒测试	模块间的接口以及模块组合后的整体功能	接口覆盖率
系统测试	黑盒测试	整个系统对需求的符合度	测试用例对需求的覆盖率

对于不同的测试需求和行业需求，Python 提供了不同的测试工具。

- unittest：junit 的 Python 版本，支持 test fixture、test case、test suite 和 test runner。
- doctest：将 docstring 作为测试范例的框架，查找和执行交互式 Python 会话，执行并检验其符合预期。
- py.test：符合 Python 风格的习惯做法，让开发人员能够以非常紧凑的风格编写测试套件。
- nose：支持与 py.test 相同的测试习惯做法，但是这个包更容易安装和维护。
- zope.testing：Zope 工程中的测试框架。支持 unittest/doctest 等传统 Python 测试风格。

- coverage：Python 的覆盖测试工具。
- Selenium2：ThoughtWorks 公司的一个 Web 集成测试的强大工具。和其他的自动化工具相比，其最主要的特色是跨平台、跨浏览器。
- splinter：使用 Python 开发的开源 Web 应用测试工具。它可以自动浏览网站和模拟用户交互。对已有的自动化工具（如：Selenium、PhantomJS 和 zope.testbrowser）进行抽象，形成一个全新的上层应用 API，它使得 Web 应用编写自动化测试脚本变得更容易。
- 其他的还有 subunit、testrepository、mox、mock、fixtures、discover 等各种测试工具包。

8.5.1 unittest 单元测试

Python unittest 是用于软件单元测试的框架，可以帮助实现高质量的代码。前面提到的硬件设计领域的许多 Python 工具就看中了 Python 单元测试能力，并将其用于硬件的功能测试中。

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

单元测试用例需要仔细设计规划。

8.5.2 socket 压力测试

开发者开发联网应用，自然希望能够支持尽可能多的设备连接；而测试设备云并发连接能力的工具属于压力测试（又称冒烟测试）范畴。通过压力测试，开发者可以充分了解在系统正常工作范围内能够支持的最大连接数。单台服务器连接数超过其最大连接数时，设备云会出现 CPU 占比和存储器占比过高、进程崩溃、持久层数据遗失、操作系统杀掉进程等各种问题。

笔者测试后发现，采用 CPython 的设计，CPU 占比会比较高，采用 PyPy 加速后会改善。同时，开发者可以利用 C 语言开发的 Ngnix、mosquitto 作为连接前端，多个 Python 进程作为工作进程，实现并发连接数最大化和负载均衡。具体连接数能够支持多少，和负载均衡、MQTT 前端设计能力以及运行主机的配置有关联。Ngnix 的并发连接数是 50 万到 100 万个连接，而 mosquitto 可以实现 2 万个连接，可以满足大部分的物联网产品需求。配合 PaaS 供应商提供 IoT PaaS 和负载均衡，可以实现弹性扩展。

其实针对 Web 的压力测试（压测）工具有很多，比如 Apache 的 ab（Apache Benchmark）。Python 领域里也存在着大量的压力测试工具，但专门针对套接字的压力测试工具相对较少。以下是笔者检索到的压力测试工具：

- Python Web Performance Tool：http://pywebperf.sourceforge.net/；
- Pylot：http://www.pylot.org/gettingstarted.html；
- Pymeter：http://pymeter.sourceforge.net/。

如果读者对于单台机器做压测的结果有怀疑，那么可以利用云计算供应商的压测服务，通过多台服务器进行压力测试，这样可以获得更加真实的数据。

8.5.3　urllib2 远程记录

在物联网的开发过程中，除了常规的压力测试，还需要对 Web API 做一些数据完整性测试。比如长时间测试一个 Web API，以测试是否有数据遗漏，记录并转换为其他格式以供用户分析使用。由于 Web API 大多数是基于 HTTP 的 REST API，并以 JSON/XML 格式为主，因此采用 urllib2 来构建 HTTP 客户端可以很容易地实现这个测试目的。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

'''
    rtDataLogger.py
    A python client to cache realtime data via remote web API
    start only from 20:00 to 08:00 of next day.
'''
__author__ = 'Allan K Liu'

import urllib2
import os, sys
import datetime, time
import json
from StringIO import StringIO

start = 20
stop = 9
```

```
webapi = "http://yoururl"
devsnr = 'ABCD70013'

class webclient(object):
    def __init__(self, uri, dev):
        self.prev = None
        self.resp = None
        self.uri = uri
        self.dev = dev
        self.online = False
        self.nwk = False
        self.logfile = "realtime.json"
        self.csvfile = "realtime.csv"
        self.fp = None
        self.timeout = 5
        self.recqueue = []

    def getWebAPI(self):
        webapi = self.uri+self.dev
        try:
            resp = urllib2.urlopen(webapi, timeout=self.timeout)
        except urllib2.URLError,e:
            print "Network Error: %s"%(e.reason)
            self.online = False
            #sys.exit(-1)
            return

        self.resp = resp.read()
        io = StringIO(self.resp)
        _obj = json.load(io)

        if len(_obj['realtime']):
            self.online = True
            r = self.getCSV(_obj)
            print r
            self.writeLogfile(r)
        else:
            self.writeLogfile('.')
            self.online = False


    def getCSV(self, obj):
        f = ''
        #print len(obj['realtime'])
        for dat in obj['realtime']:
            f += "t:%s, %s\r\n"%(dat['time'], self.tsToStr(dat['time']))
            f += "p:%s\r\n"%(repr(dat['press']))
            f += "f:%s\r\n"%(repr(dat['flow']))
```

```
            f += "s:%s\r\n"%(repr(dat['sao2']))
            f += "k:%s\r\n"%(repr(dat['kpr']))
            f += "\r\n"
        return f

    def setDev(self, dev):
        self.dev = dev

    def setLogfile(self, fp=None):
        self.logfile = fp
        try:
            fp = open(self.logfile, 'a')
            fp.write('Realtime logging for %s starts here.\r\n'%(self.dev))
            fp.write('='*80 + '\r\n')
            fp.close()
        except IOError:
            print "File IO Error: %s"%(IOError.e)

    def writeLogfile(self, cxt):
        try:
            fp = open(self.logfile, 'a')
            fp.write(cxt)
            fp.close()
        except IOError:
            print "File IO Error: %s"%(IOError.e)

    def isOnline(self):
        return self.online

    def tsToStr(self, ts):
        i,f = ts.split('.')
        ta = time.localtime(int(i))
        timestr = time.strftime("%Y-%m-%d %H:%M:%S", ta)
        timestr = timestr + '.' + f
        return timestr


def realTimeDataLogger():
    global start, stop, webapi

    print("Realtime Data Remote Logger")
    print("by Allan K Liu\r\nCtrl-C to stop\r\n")

    logger = webclient(uri=webapi, dev=devsnr)
    logger.setLogfile('IoT_realtime.log.txt')

    while 1:
        #_hour = datetime.datetime.now().hour
```

```
            logger.getWebAPI()
            if logger.online:
                time.sleep(2.75)
            else:
                print "Sleep %d seconds due to NWK error or Device is offline."
                time.sleep(60)

if __name__=='__main__':
    realTimeDataLogger()
```

以上例子，可以简单理解为针对测试目的而设计的 Web API 的爬虫。套接字服务器一侧可以使用规则波形如三角波、正弦波作为激励信号，这样绘图后可以很明显地发现是否有数据丢失。

8.5.4 PCBA 测试

PCB 焊接后就是 PCBA（PCB Assembly）。PCBA 生产线最后需要设置测试工位，以检出虚焊的 PCBA 并返工。批量生产需求是简单、直观、快速，所以需要定制一些特殊设备。

专注于开源硬件的美国 Sparkfun 公司针对电路板自动测试提供了开源测试台。测试台组合完毕后的效果图可参见图 8-2。这方面的制作过程可以参考 Sparkfun 公司的博客：《如何利用测试针（Pogo Pin）来做测试台》。该文介绍了使用测试针来保持测试台与 PCBA 测试点的电气连接，并利用转换 IC/MCU 保持与主机通信，同时通过 LED 或者终端打印的方式来显示对应引脚的焊接情况。该方法支持模拟和数字引脚功能。

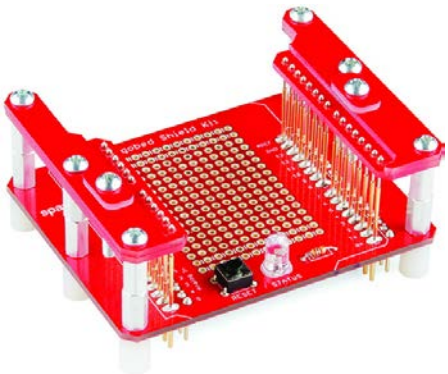


图 8-2　Sparkfun 公司设计的电路板测试台

在焊接完所有元器件之后，需要对板上所有外设的基础功能进行测试。如果 PCB 上是 MCU 应用板，那么需要对 MCU、片内外设（GPIO、ADC、PWM、I2C、SPI）和板载外设进行测试。通常情况下，批量生产的 PCBA 需要专门的测试台，以及测试固件和测试软件。

市场上的各测试针结构类似，并形成了工业标准。图 8-3 为一种测试针（Pogo Pin）放大图。该针体为铜质，针内有细小弹簧，针体焊接在测试台上，将待测电路板（DUT，Device Under Test）压在测试针针头上，实现与测试台之间稳定的电气连接。由于 PCB 焊盘的多样性，因此测试针的针头有许多形状：尖头、圆头、平头、爪形，具体型号请检索供应商与电商市场的数据。

图 8-3　工业标准测试针

8.5.4.1　测试台

测试台由 RS232/USB、电源、LED 显示和键盘构成，PCB 上还有测试针。测试工位的工人只需要将 DUT 压入测试台，不需要焊接就可以完成系统连接。

8.5.4.2　测试固件

由于测试台通过 USB 连接到 PC，因此可以通过测试针将测试固件下载到 DUT 主控 MCU 中。该固件的功能就是根据 PC 主机命令去运行测试用例，如切换 GPIO（LED 闪烁）、PWM 输出（LED 亮度变换）、ADC 采样（PC 终端屏显）等。

8.5.4.3　测试软件

测试软件运行于测试主机，发送高层命令切换不同的测试模式。DUT 和主机之间可以采用 UART/USB/socket 连接。主机端可以通过 pyserial/socket 与 DUT 保持通信进行自动测试。

8.5.4.4　GPIO 测试

PCBA 测试主要测试焊接情况，所有引脚测试都可以最终落实到 GPIO 测试，而无须考虑 GPIO 的第二功能特性。当测试固件切换 GPIO 高低电平时，LED 可以显示出对应引脚的状态和焊接状态。

8.5.4.5 ADC 测试

测试模拟输入时，可以不断轮询 ADC 的所有通道，测试人员可以用手或电容笔触摸 ADC 的引脚。主机测试软件可以将 ADC 采样数据以棒状图显示，正常情况下可以看到 ADC 的波形在不断地变化。其运行效果如图 8-4 所示。这种测试波形精度比较粗糙，时间轴是纵向的，但用于测试 ADC 采样工作是否正常却足够了。其主要用于排查短路、断路或者其他外电路故障。

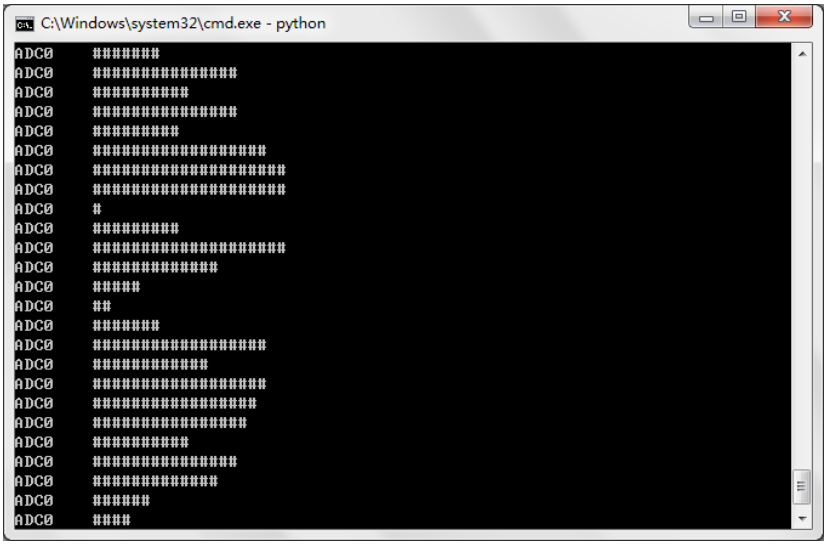


图 8-4　测试台 ADC 测试终端

利用 Python 配合测试台可以很简单地完成测试自动化。测试完毕后，还可以继续进行固件更新、个性化与加密、设备串号登记入库等后续操作。UART/USB 串口上的应用通信协议，可以参考供应商的 Bootloader 协议和 Arduino/mbed 的相关协议。

8.6　文档生成器

软件工程师对于文档总抱有复杂的情绪。在理论上和工程实践中，他们知道文档是一个非常关键的软件质量管理的控制项目；但他们又感觉文档占据了自己太多的工作时间，每改一处代码，需要更新多种文档，并抄送给相关人员，这实在是一件让人纠结的事情。

所以，工程师们会非常自然地想到：是否可以尝试在代码中嵌入文档，利用某种工具软件来分析源码，提取、生成文档并自动更新所有相关文档？至少自动生成文档是可以做到的，而自动更新所有文档并推送给相关人员则需要更多网络服务配合。

如果文档中有 HTML 相关的格式，如 Markdown/reST/XML 之类的，可以采用配置工具，即版本控制工具进行管理。这种方式更加适合具有程序员背景的人来构建，并导出静态网站。

8.6.1　文档格式

在文档自动生成流程中，需要先从代码中提取文档，合并文本，并转换到对应的文档格式。然而在这三个环节中可能需要不同的格式。首先，代码中的文档要有合适的表达或格式，以区别于代码中普通的注释。其次，纯文本也需要有合适的格式。作为抓取结果的中间结果可能也需要一种格式。最后要形成的格式有可能千差万别：HTML、cHTML、PDF、ePub 或 LaTeX。

8.6.1.1　docstring

让程序员在源码之外单独编写文档容易出现文档与代码实现不同步的问题，那么从源码及注释文字中提取文档算得上是源码文档同步的方法之一了。docstring，即文档字符串在许多语言中都得到了支持，如 Lisp、Python、Cloujure、Julia。doscstring 是 Python 自带的功能，所有 Python 模块文档字符串都可以被 help 所提取，并通过__doc__方法调用。

在 Python 代码中，紧跟着‘def’和‘class’语句，即函数或类定义后的第一个逻辑行字符串就是该函数或者类的文档字符串。如果是模块的文档字符串，则需要放置在文件的第一行。文档字符串可以是一行，也可以是多行。

Python 源码：mymodule.py

```
"""
Assuming this is file mymodule.py, then this string, being the
first statement in the file, will become the "mymodule" module's
docstring when the file is imported.
"""

class MyClass(object):
    """The class's docstring"""
    pass

    def my_method(self):
        """The method's docstring"""
        pass

def my_function():
    """The function's docstring"""
    pass
```

在 REPL 中查看 docstring：

```
>>> import mymodule
>>> help(mymodule)
```

```
Assuming this is file mymodule.py then this string, being the
first statement in the file will become the mymodule modules
docstring when the file is imported

>>> help(mymodule.MyClass)
The class's docstring
>>> help(mymodule.MyClass.my_method)
The method's docstring
>>> help(mymodule.my_function)
The function's docstring
>>>
```

Python docstring 还可以导入 pydoc 和 Sphinx，形成复杂的多层级项目文档。

8.6.1.2 PythonDoc

PythonDoc 是 JavaDoc 的 Python 实现。该工具会在 Python 源代码的注释文本中提取特殊的文档字符串，来获取对于变量、函数、类、方法的说明文档。PythonDoc 注解的第一行必须使用##，之后的每行必须采用#。PythonDoc 会将#符号以及随后的空格删除。其他的空格保留。

Pythondoc 支持的标签如下：

```
@param，参数
@keyparam，关键字参数
@return，返回值
@exception，异常
@def，默认值
@link，内联连接
@linkplain，HTML 内联连接
```

Pythondoc 示例如下：

```
##
# This comment provides documentation for the following
# function.
# @param size The requested size, as an integer.
# @param filename The source file.  Instead of a file name, you can
#     provide a file object, or any other object that supports 'read'
#     and 'seek'.
# @return An integer value.

def function(argument, size, filename):
    return size*100

##
# This comment provides documentation for the following
# variable.
```

```
variable = value
```

pythondoc.py 命令行工具可以导出 HTML、XML，或通过定制解析器导出其他格式。

```
$ pythondoc.py package
$ pythondoc.py module1.py module2.py module3.py package/module4.py
$ pythondoc.py -x module.py
$ pythondoc.py -Omyformat -Dindex -Dencoding=utf-8 mypackage
```

更多详情可查看本章延伸阅读部分中有关 effbot 的 Pythondoc 说明。

8.6.1.3 Markdown

HTML 的全称是 HyperText Markup Language。HTML 作为 XML 的子集，虽然很实用，但依然显得不够简洁。一个 HTML 标签必须是封闭完整的，即必须以结束，否则就是不完整的标签。从字面上理解，Markdown 是与 Markup 相对而言的，是一种更加简单的语言。Markdown 依然是一种标记语言，它也有自己的简单标记语法，可以使用普通文本编辑器编写。通过这些标记语法，它可以使普通文本内容具有一定的格式。Markdown 具有一系列衍生版本，用于扩展 Markdown 的功能（如表格、脚注、内嵌 HTML 等）。这些功能在最基本的 Markdown 版本中尚不具备，但此类衍生版本能让 Markdown 转换成更多的格式，例如 LaTeX、Docbook 等格式。Markdown 在 Web 平台开始流行，并逐渐拓展到桌面和云笔记中，笔者常用的 Sublime Text2 编辑器和有道云笔记就支持 Markdown。

典型的 Markdown 例子如下：

```
# 一级标题

## 二级标题

### 三级标题

####无序列表
* abcd
* efg
* xyz

####有序列表
1. abcd
2. efg
3. xyz

####表格

| Tables        | Are           | Cool  |
```

```
| ------------- |:-------------:| -----:|
| col 3 is      | right-aligned | $1600 |
| col 2 is      | centered      |   $12 |
| zebra stripes | are neat      |    $1 |

####引用

> 举头望明月，低头思故乡

####URL
[baidu](http://www.baidu.com/)

####图片
![mypic](http://domona.com/static/pix.png)
```

表格是 Markdown 中最薄弱的一环，居然要手动填充空格才能够形成表格。

8.6.1.4 reST

结构化文本（StructureText）的初衷是避免使用 TeX、Troff 或者 HTML/SGML/XML 等标记语言中的各种标记。其后衍生了若干版本，包括：

- StructuredTextNG；
- Setext；
- PyTextile，用于 Web；
- reStructuredText。

reStrucuturedText 即 reST，是 David Goodger 重新编写的 StructureText（结构化文本）包，re 即重写的意思。reST 主要用于文档目的。reST 作为 Python Docutils 包的组成部分，其成为主流的 Python 文档格式之一，它的后缀名为*.rst。最常用的方式就是将 reST 格式文件转换成 HTML 网页文档用于在线文档分享。

reST 示例如下：

```
*强调*
**重点强调**
``代码示例``

* 无序列表 1
* 无序列表 2

1. 有序列表 1
2. 有序列表 2

#. 有序列表 1
```

```
#. 有序列表 2

+------------------------+------------+----------+----------+
| Header row, column 1   | Header 2   | Header 3 | Header 4 |
| (header rows optional) |            |          |          |
+========================+============+==========+==========+
| body row 1, column 1   | column 2   | column 3 | column 4 |
+------------------------+------------+----------+----------+
| body row 2             | ...        | ...      |          |
+------------------------+------------+----------+----------+

=====  =====  =======
A      B      A and B
=====  =====  =======
False  False  False
True   False  False
False  True   False
True   True   True
=====  =====  =======

=================
标题
=================

"#"针对篇
"*"针对章
"="针对节
"-"针对目
"^"针对子目
"""针对更细的分支

.. csv-table:: sample table
    :header: "A","B","C"
    :widths: 20,20,20

    "X","Y","Z"
    "1","2","3"
```

从语法角度来看，reST 与 Markdown 有类似的地方。但是 reST 入门要比 Markdown 稍难一些，其更适合集中编写文档，然后一次性转换成 HTML 或其他格式。本书编写时就采用了 reST/Sphinx 格式。

8.6.2　文档生成工具

接下来介绍一下如何将这些原始文档转换成其他格式的工具和服务。

8.6.2.1 Sphinx

Sphinx 是一种文档生成工具，依赖于 pygments、docutils、babel、snowballstemmer 等许多其他 Python 软件包。其徽标如图 8-5 所示。Sphinx 可以从 reST 源文件中产生文档，还可以使用 autodoc 扩展产生 API 页面。Sphinx 是可扩展的，支持多种输出格式，并为生成的 HTML 文件定制不同主题。Sphinx 内置对于文档的静态检索，可以在文档源码中检索关键字，并生成新的网页。

图 8-5 Sphinx 徽标

本书一开始采用云笔记编写。在本书编写之初，云笔记仅支持 HTML 格式在线编辑。由于书稿内容的增长，已经超过云笔记的编辑能力，因此笔者中途不得不切换成 reST/Sphinx 来编写本书。此外为了版本控制，笔者使用了 Git 进行管理。将 reST 格式内容放置在自家服务器中。离线编辑后与服务器同步，或者登录服务器编辑均可。Sphinx 在将 reST 源文件转换成 HTML/ePub 的过程中，使用 makefile 来产生目标格式，这类似于“软件编译”。这非常符合软件工程师的使用习惯。

笔者写作本书主要使用了 reST/Sphinx 等 Python 相关技术，再配合使用 Markdown 进行了部分内容的编辑。笔者使用体验良好，对于文本、代码、图片、索引都很满意。但无论是 Markdown 还是 reST，原始格式中的表格均需要手动填充空格，实在烦琐；使用 CSV 表格输入比较合理。好在 Sphinx 支持 csv-table，可以解决此问题。

采用 Sphinx 后，笔者可以重点关注于写作内容，而出版社可以利用 CSS 和其他转换器生成桌面和电子出版所需的格式。Sphinx 的使用也很简单，运行一个启动脚本，它会通过向用户提一些问题来构建项目的 conf.py 配置文件，给出一个空白工程文件 index.rst，剩下的由用户填充、编写内容，并通过 makefile 来构建输出文档。该启动脚本如下：

```
$ sphinx-quickstart
```

此外，笔者发现 Sphinx 在 Linux 中明显运行得更快，虽然转换命令在 Windows/Linux 的用法是一致的。

```
make html
make epub
```

但在 Linux 中，make 调用的是 gmake 和 makefile。在 Windows 中调用的是 make.bat 文件和对应的 Sphinx Python 脚本。

8.6.2.2　Doxygen

和 docstring 工作原理类似，Doxygen 可以从多种编程语言源码中提取注释。它是一种覆盖率比较好的工具。其目前可以支持的语言如下：

- C/C++/C#/Objective-C；
- PHP；
- Java；
- Python；
- IDL；
- Fortran；
- VHDL；
- 其他语法类似的语言。

通过编写过滤器，Doxygen 可以继续支持新的编程语言和注释方式。Doxygen 同时支持两种风格的 Python 注释。在实际使用中，其需要与 DoxPyPy 过滤器配合使用。Python 的源码文档提取得到了 dostring/PythonDoc 等许多工具的支持，笔者关注的是 Doxygen 如何从其他语言如 C/C++/C#/Java 中提出文档。

在 Doxygen 的文档中详细描述了在 C/C++/Java 中如何标注以下对象。

- 文件头标注：@file、@brief、@author、@email、@version、@date、@license；
- 命名空间标注；
- 类、结构、枚举标注；
- 函数注释原则：@brief、@param、@return、@note、@retval、@pre 等；
- 变量注释；
- 模块标注：@name；
- 分组标注：@name。

由于支持多种语言文档，因此 Doxygen 的语法看上去比较复杂，需要仔细研读其文档后再实施。

8.6.2.3　MkDocs

MkDocs 可用于创建简单、快速的项目文档静态站点。文档源码使用 Markdown 来撰写，用 YAML 文件作为配置文档。MkDocs 的特点如下：

- 易于托管。由于采用静态 HTML 网页构成，因此其可以被托管到任意服务器中。
- 主题丰富。可以采用 bootstrap、readthedocs 和 12 款 bootswatch 主题。
- 即时预览。内置开发服务器，刷新网页即可以预览。
- 易于配置。文档主题配置采用 YAML。

- 交叉索引。使用 MkDocs 连接语言建立交叉索引。

8.6.2.4 Pycco

Pycco 是 Docco 的 Python 版本。Docco 是一个应急用的文档生成器，使用 Literate CoffeeScript 编写。使用 Markdown 语言，可生成 HTML 文档来显示代码中的注释，代码通过 Pygments 语法高亮。

8.6.2.5 Pandoc

不得不说，不同文档的格式转换是一件“脏活”。在这方面有无数的软件尝试过，但是都做得不是很理想。笔者推荐使用 Pandoc 作为一种终极文档转换工具，该软件用 Haskell 语言开发。该软件支持的源格式和目标格式如下：

Markdown、reStructuredText、textile、HTML、DocBook、LaTeX、MediaWiki markup、TWiki markup、OPML、Emacs Org-Mode、Txt2Tags、Microsoft Word docx、LibreOffice ODT、EPUB、以及 Haddock markup。

安装如下：

```
$ sudo apt-get install pandoc
```

命令行语法如下：

```
$ pandoc -o myword.docx myword.rst
```

利用以上语句，可以将笔者所写的所有 reST 格式书稿转换为交付给出版社的 docx 文件，不带任何格式。简单即美！

Pandoc 维护者 John MacFarlane 非常努力，在论坛中反馈得也很及时。但其依然有不少 bug。即便有源码，Haskell 代码也不是普通开发者能够很快入门的。在此有一些建议仅供读者参考：

- 推荐在 64 位操作系统上安装 Pandoc，包括 Windows 和 Linux。Pandoc 默认不提供 32 位操作系统支持。
- 推荐及时更新，以减少 bug。
- Pandoc 出错分为 reader 和 writer 两个部分，可以使用“pandoc -o file.native sourcefile.ext”来定位是前端还是后端问题。
- 如果直接转换路径不行，可以利用中间格式。比如，笔者发现 reST 转 docx 会出现丢失插图和表格的问题，而利用 HTML 做中间格式转换就没有任何问题。

真是条条大路通罗马啊！

8.7　文档操纵

除了在线出版可以使用 HTML、PDF 格式，纸媒出版需要 LaTeX 外，大多数情况下我们依然需要使用主流的 Office 文件格式：Word 的 docx 和 Excel 的 xlsx。

8.7.1　Doc 文档操纵

python-docx 是 Python OpenXML 的一部分，其用于创建和更新 Microsoft Word（.docx），即 Word 2007 及以后的文档。

安装如下：

```
pip install python-docx
```

python-docx 演示代码如下：

```
from docx import Document
from docx.shared import Inches

document = Document()

document.add_heading('Document Title', 0)

p = document.add_paragraph('A plain paragraph having some ')
p.add_run('bold').bold = True
p.add_run(' and some ')
p.add_run('italic.').italic = True

document.add_heading('Heading, level 1', level=1)
document.add_paragraph('Intense quote', style='IntenseQuote')

document.add_paragraph(
    'first item in unordered list', style='ListBullet'
)
document.add_paragraph(
    'first item in ordered list', style='ListNumber'
)

document.add_picture('monty-truth.png', width=Inches(1.25))

table = document.add_table(rows=1, cols=3)
hdr_cells = table.rows[0].cells
hdr_cells[0].text = 'Qty'
hdr_cells[1].text = 'Id'
hdr_cells[2].text = 'Desc'
for item in recordset:
```

```
        row_cells = table.add_row().cells
        row_cells[0].text = str(item.qty)
        row_cells[1].text = str(item.id)
        row_cells[2].text = item.desc

    document.add_page_break()

    document.save('demo.docx')
```

8.7.2 Excel 表格操纵

在实际的工程控制如软件质量管理活动中，Excel 的使用率甚至超过 Microsoft Project 的甘特图。此外，Excel 以及 CSV 格式文档也是互联网和物联网运维甚至用户数据批量上传的手段。因为 Excel 的受众更多，所以 Python 有一些 Excel 的读/写模块，其中包括标准库中的 csv 模块。还有第三方 xlrd、xlwt 模块。前者为读取操作，后者为写入操作。这几种常见的 Python Excel 模块各方面对比可参见表 8-2。

表 8-2 Python Excel 模块对比表

名　称	功　能	速　度	操作系统	备　注
XlsxWriter	强	快	N/A	仅创建 xlsx，跨平台，复杂功能
python-excel	弱	快	N/A	读/写 xlsx 及早期版本 xls，跨平台，简单功能
OpenPyXL	中等	快	N/A	持续修改 xlsx，跨平台，功能复杂
COM API	最强	慢	Windows+Excel	仅适用于 Windows，各种格式

以上对比数据来自一篇网络博文《用 Python 读写 Excel 文件总结》，网址可查看本章延伸阅读部分。

在软件质量管理中，可以通过 Python 配合 Git/SVN 以及私有源码版本管理工具来检查源码修改情况，合并测试报告，更新 Excel 文件。

Excel 软件广泛用于日常统计分析报表中，混杂了数据、公式、脚本和图表，所以它不能够简单地以数据库来替代。由于第三方模块不能够完整地掌握不同版本的 Excel 格式，因此程序化更新 Excel 文件有数据丢失的风险。读取和创建 Excel 的风险小于更新 Excel 文档。

在大多数情况下，Excel 也是以文件方式在团队内部流转的，这容易造成版本不一致的问题。也正因为如此，笔者个人倾向于在统计系统建立之初，就将数据单独保存在数据库中，将公式和脚本用代码实现的方式来规划。Excel 可以作为一个大家都可以使用的数据统计前端而存在，从服务器和数据库获取数据源。

8.8　国际化与本地化

中国已经深度融入了国际市场，产品国际化和本地化是必须要做到的。那么如何定义国际化和本地化呢？

- 国际化，Internationalization（i18n），使产品设计能够同时满足多个市场的需求，而无须重新设计系统程序的工程方法；
- 本地化，Localization（l10n），使得产品能够满足特定区域市场需求的设计方法。

前者是将应用系统中的程序代码与语言和文化相关的字体、字符串、媒体、格式等资源分离，这样升级资源文件并不会影响到系统代码的实现。后者是在国际化基础上，针对特定文化圈需求进行小规模定制和适配。

国际化和本地化是同一件事情的两面。对于北美设计师来说，国际化和本地化是一个强制需求。几乎所有产品都必须支持四种语言：英语、法语、西班牙语和葡萄牙语。对于中国设计师来说，首先，要保证一个国际化的框架，不要把资源固化在系统程序中；其次，使用英语而非中文作为默认语言填充所有资源内容；最后，再针对不同语言背景增加各个本地化的设计。

8.8.1　gettext

gettext 是 GNU 翻译工程的组成部分。该软件包为程序员、译者及用户提供了完整的工具和文档。尤其是，GNU gettext 工具为帮助其他 GNU 软件包实现多语种消息而提供了一整套的工具。这些工具包括：

- 程序应该如何编写才可以支持消息目录；
- 为消息目录准备目录和组织文件名称；
- 用于抽取翻译的消息运行时的库；
- 若干独立程序用于处理可翻译的字符串、已翻译的字符串等；
- 还有一个特殊的 GNU Emacs 模式用于准备并更新这些字符集。

在实际工程中，gettext 被大量工程所使用，在命令行中输入：

```
xgettext --help

Choice of input file language:
  -L, --language=NAME        recognise the specified language
                              (C, C++, ObjectiveC, PO, Shell, Python, Lisp,
                              EmacsLisp, librep, Scheme, Smalltalk, Java,
                              JavaProperties, C#, awk, YCP, Tcl, Perl, PHP,
                              GCC-source, NXStringTable, RST, Glade)
  -C, --c++                  shorthand for --language=C++
By default the language is guessed depending on the input file name extension.
```

大多数编程语言均可以受益于 gettext 项目。这体现在实际商品，如消费电子产品中，也可

以体现在 Web 设计界面中。笔者以 Python 中的一个 Web 框架 Cyclone Web（作者 Alexandre Fiori）的国际化和本地化流程为例做简单介绍。

Cyclone 在设计中就实现了国际化框架，并利用了 GNU 标准工具集来实现国际化和本地化。其流程如下：

（1）采用 xgettext，从源码中提取并产生 pot 翻译文件；

（2）如果提取字符串失败，采用不同方式和 Python 工具来寻找字符串；

（3）采用 msgmerge，将新产生的 pot 文件与旧的 pot 文件合并成新的 pot 文件；

（4）人工翻译 pot 文件中的字符串；

（5）采用 msgfmt，将 pot 文件编译为 mo 文件；

（6）将翻译好的 mo 文件放置在指定位置。

从“mytest”产生 pot 翻译文件：

```
xgettext --language=Python --keyword=_:1,2 -d mytest localedemo.py \
    frontend/template/*
```

在 Web 模板中使用翻译字符串时，变量参数必须使用_()函数包裹起来。例如：

```
_("foobar %s" % x) # wrong
_("foobar %s") % x # correct
```

xgettext 在解析某些 HTML 元素时可能会失效，遗漏某些字符串，例如：

```
<input type="submit" value="{{ _('bla') }}" />
```

那么可以尝试 xgettext –language=Php 选项，或将其导向另外一个工具脚本 localefix.py：

```
#!/usr/bin/env python
# localefix.py
import re
import sys

if __name__ == "__main__":
    try:
        filename = sys.argv[1]
        assert filename != "-"
        fd = open(filename)
    except:
        fd = sys.stdin

    line_re = re.compile(r'="([^"]+)"')
    for line in fd:
        line = line_re.sub(r"=\\1", line)
        sys.stdout.write(line)
    fd.close()
```

此时收集到的 pot 文件内容如下：

```
# cyclone locale demo
# Copyright (C) 2010 Alexandre Fiori
# This file is distributed under the same license as the cyclone package.
# Alexandre Fiori <fiorix@gmail.com>, 2010
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: 0.1\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2010-01-20 12:19-0200\n"
"PO-Revision-Date: 2010-01-20 12:21-0300\n"
"Last-Translator: Alexandre Fiori <fiorix@gmail.com>\n"
"Language-Team: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=INTEGER; plural=n != 1;\n"

#: localedemo.py:76
msgid "hello world"
msgstr "hola mundo"

#: frontend/template/hello.txt:2
#, python-format
msgid "%s, translated"
msgstr "%s, traducido"

#: frontend/template/index.html:5 frontend/template/index.html:25
msgid "cyclone locale demo"
msgstr "demonstración de internacionalización de cyclone"

#: frontend/template/index.html:29
msgid "Please select your language:"
msgstr "Por favor, seleccione su idioma:"

#: frontend/template/index.html:35
msgid "How many apples?"
msgstr "Cuantas manzanas?"

#: frontend/template/index.html:39
msgid "Submit"
msgstr "Enviar"
```

其中 msgid 是所有字符串索引；而 msgstr 在初始状态下是空白的，需要人工翻译填充。上

面是西班牙语的例子。

将翻译后的 pot 文件与原有的 pot 文件合并起来：

```
msgmerge old.po mytest.po > new.po
mv new.po mytest.po
```

将 pot 文件编译成 mo 文件：

```
msgfmt mytest.po -o locale/{lang}/LC_MESSAGES/mytest.mo
```

类似地，可以将各个语种的 pot 文件编译并复制到 Web 框架规定的文件夹下：

```
msgfmt mytest_pt_BR.po -o frontend/locale/pt_BR/LC_MESSAGES/mytest.mo
msgfmt mytest_es_ES.po -o frontend/locale/es_ES/LC_MESSAGES/mytest.mo
```

接下来我们看看 Web 框架如何实现多语种切换。

8.8.2 Web 多语种切换

Web 多语种设计主要体现在 Web GUI 中常见字符串（比如 Submit、OK、SignIn/SignOut）的自动切换，而不是 Web 内容方面。Web 内容应该针对特定地区专门编写，比如中国境内的“双 11”购物节促销对于美国的购物者而言是无意义的，就必须是中文内容。

多语种信息与 IP 地址、用户偏爱有关，可以动态监测或保存在 Cookie 和数据库中，也可以用 HTTP 参数来强制，例如：

```
http://yourdomain.com/language=zh?
http://yourdomain.com/language=fr?
```

Web 应用程序应当从 URL 参数、IP 地址、Cookie 和数据库各个参数综合判断出最终显示的语言，从 mo 文件中提取出最终的字符串显示在网页中。在 Cyclone 的 locale.py 中可以看到一个成熟框架的做法：除了语种的判断、字符串的提取，该文件中还有关于日期显示以及数字方面的差异。

8.8.3 字库文件生成器

字库提取是嵌入式开发中常见的与 GUI 有关联的任务。由于嵌入式系统中显示设备的扫描方式和标准显示设备如 VGA 的扫描方式不同，所以需要进行字库提取和转换。

8.8.4 GB2312 点阵字库提取

早前开发电视机、显示器等，就需要在 OSD ROM 中烧录各种点阵字符。此外，具备 LCD/EPD/VFD/OLED 等显示器件的设备开发，以及 POS 机中的微型打印机也是基于点阵字库

的，也需要提取字模，形成字库。

在早期计算机的汉字系统中，UCDOS 是一个必须提及的操作系统。UCDOS 给我们留下了宝贵的遗产：GB2312 汉字字库。GB2312 字库一般都是点阵字库，基础是 HZK16（16x16）的仿宋体字库，其后又有了 HZK12、HZK24、HZK32、HZK48，还衍生出黑体、楷体、隶书。其中，部分汉字字库字体扫描方式与其他有所不同。其提取出来后很适合用于嵌入式系统的开发。这个原理同样适用于其他国家语言的点阵字库。大多数国家的语言多采用字母，如拉丁语系，以及越南文、韩文和日文假名都可以提取独立字母的字模，汉字也是可以独立显示的。

采用 Python 提取 GB2312 字库很简单，读取 HZK 文件中的二进制字模，并将其转换成自己所需要的扫描方式和大小端即可，但这与 HZK 中的扫描方式和目标显示硬件扫描方式有关联。前面提到有以下三个维度。

- 扫描方位：水平或垂直，确定了字体是站着的还是躺着的；
- 扫描方向：正向或反向，确定了字体前后朝向和上下朝向；
- 数据存储：大端或小端，确定了字体内部存储方式。

一些显示器件有些特殊要求，如 8.10.1 节中介绍的电子墨水屏模块，其灰度是 2 位的，这个字模到最终显示过程中还需要做一次灰度转换。另外，笔者还遇到过显像管电视 OSD 在纵横两个方向上不一致，也需要转换的情况。这些设备的显示驱动都需要有针对性的定制。

8.8.5　TTF 字库提取

点阵字体往往依赖于显示密度和面积，无法进行缩放。如果要产生边界平滑而可以无级缩放的字体，需要用到各类矢量字体。可以针对特定的显示属性，将适量字体放大后转换为点阵字体后提取出来。以下是各类型的矢量字体。

1．TrueType（.ttf）

这是 Windows/Mac 系统常用的字体格式。Apple 公司拥有涉及 TrueType 字形轮廓（glyph outline）处理方面的三项专利，分别于 1989 年和 1992 年申请。这些专利并没有阻止任何人读取、转换或者生成 TrueType 字库。它们仅仅聚焦于 hinting TrueType 字形时的一些微妙技巧上。只要在优化小尺寸点阵时没有涉及这些专利技术，用 TrueType 字库来显示文字就是合法的。

2．EOT，Embedded Open Type（.eot）

嵌入字体格式为微软开发的字体技术，可以将 OpenType 字体嵌入网页并下载到浏览器渲染。

3．OpenType（.otf）

采用 PostScript 格式的可缩放字体，为微软和 Adobe 联合开发。

4．WOFF，Web Open Font Format（.woff）

WOFF 是适用于网页的字体，是 W3C 推荐标准。WOFF 大体上是带压缩和附加元数据的 OpenType 和 TrueType 字体，目的是减少带宽使用。

5．SVG，Scalable Vector Graphic（*.svg）

SVG 是一种矢量图形，也可以用于描绘字体。

理论上，矢量字体可以缩放而不失真。但是如何缩放、计算和提取？在其算法上笔者也缺乏经验。于是笔者在 Stackoverflow 上提出了这个问题，有位热心网友（Aarni Koskela）用 Python 代码告诉了笔者如何提取 TTF 字体并转换为点阵字体，请查看本章延伸阅读部分。

Aarni 提供的 Python-minift 是基于 Python ctypes 接口的方案，利用 freetypes6.dll 访问字体。以下是利用该软件包从 TTF 字库中提取并转换的小点阵字体，具体代码请参考本章延伸阅读部分的网址。

在图 8-6 中，显示了从 TTF 字库中生成了 8×8 点阵字体。虽然矢量字体变成小矩阵的点阵字体依然会变形，显得较难看，但是将其放大到 16 点阵以上就显得很美观了。所以，到目前为止，最小的汉字点阵还是推荐使用 12×12。感谢 UCDOS 的贡献。

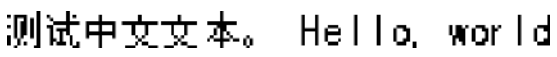


图 8-6　Python ctypes 提取 TTF 字库转为点阵字体

8.9　配置管理

配置管理（Configuration Management，CM）是通过技术或行政手段对软件产品及其开发过程和生命周期进行控制、规范的一系列措施。配置管理的目标是记录软件产品的演化过程，确保软件开发者在软件生命周期中的各个阶段都能得到精确的产品配置。

8.9.1　软件配置管理

软件配置管理（Software Configuration Management，SCM）是配置管理的重要组成部分。其中最常见的工具就是版本控制工具。需要注意的是，即便是一个人开发的项目，也有版本控制的必要。此类工具也经历了很多变化。早期，开发者主要使用 RCS、CVS，现在最常使用的是 Subversion、hg 和 Git。

- **RCS（Revision Control System）**即程序改动控制系统，其主要功能是用来管理文件的版本，可以节省空间和时间。这样就不需要在每个程序开发到某一个阶段就将数据复制到其他地方备份起来了。
- **CVS（Concurrent Versions System）** 代表协作版本系统或者并发版本系统，是一种版本控制系统，方便软件的开发和使用者协同工作。

Subversion（简称 SVN），是一个开放源代码的版本控制系统，相对于 RCS、CVS，它采

用了分支管理系统，其设计目标就是取代 CVS。曾经有一度，互联网上有许多配置管理服务从 CVS 转移了到 Subversion。

hg/mercurial，即分布式的 CVS。与传统 CVS 的区别在于，其代码仓库可以在本地，不一定放在集中服务器中。多个用户修改同一对象也不需要锁定文件。

Git 和其他版本控制系统（如 CVS）有不少的差别，Git 本身关心档案的整体性是否有改变；但多数的 CVS，或 Subversion 系统则在乎档案内容的差异。因此 Git 更像一个档案系统，可以直接在本机上取得资料，不必非得连线到主机端取回资料。

笔者的大部分项目均为一个人开发，最早使用的版本控制系统是一个免费的 RCS 软件。现在因为使用开源软件，必须使用大家都利用的工具，所以常常使用 Git 和 Subversion。

8.9.2　软件配置管理自动化

软件配置一般是人工操作，但是为什么要自动化呢？软件配置管理自动化可以为配置管理工具增加监督检查、超时提醒功能：在项目初期根据开发计划、测试计划、配置管理计划设定文档、代码的提交日期，并在指定日期对于事件是否发生予以检查，并提醒事件负责人和配置管理员做出相关处理。还可以为配置管理工具增加自动备份功能，以及为配置管理工具增加流程控制功能。当然，也可以用于实现自动化管理与持续交付。

软件配置管理自动化软件包 Gittle 的作者列举了他开发的目的：

- Git repo 管理自动化（push、pull、commit 等指令）；
- 利用 Python 的脚本化，可以从 Python 中调用；
- 在 SOA 环境中实现易用目的和更好的可操作性。

只有重度依赖配置管理工具的软件工程师、测试工程师、运维工程师和质量工程师才能够充分体会软件配置自动化的好处。在 ARM mbed 提供的工具箱中，已经开始在生产线上利用 Python 实现自动测试以及固件的自动化软件配置管理。

8.9.3　Git Bash

Git Bash 是命令行工具，其支持 MINGW32 等环境。Python 可以利用 os 模块来调用 Git Bash 的各类命令。这应该是最简单的方式。

```
import os
os.system('git')
```

8.9.4　Dulwich/Gittle 包

Dulwich 是一个 Git 文件格式和协议的纯 Python 实现，支持读取资源库内容、索引以及通

过 Git 网络协议进行数据的读/写，支持本地和远程资料库。Gittle 是一个基于 Dulwich 的高级 Git 库，完全用 Python 实现，接口使用起来十分简单。

```
from gittle import Gittle

repo_path = '/tmp/gittle_bare'
repo_url = 'git://github.com/FriendCode/gittle.git'

auth = GittleAuth(pkey=key)
repo = Gittle.clone(repo_url, repo_path, auth=auth)
```

8.9.5 Python Subversion 包

Debian 中收录了 Python 中常见的 Subversion 包，列表如下：

- python-svn——Subversion 的 Python 接口包；
- python-subversion——Subversion 的 Python 绑定；
- python-subvertpy——另外一种 Subversion 的 Python 绑定。

8.9.6 watchdog 系统监控

watchdog 是监控特定文件是否被修改的一个 Python 扩展包，一般在系统运维中使用。其可以用于监控源文件是否被修改，并进一步通过预先写好的策略来确定是否需要将代码更新到代码仓库中去或者通知相关负责人员。

8.10 数据与素材处理

在许多工程中，对于初级数据进行整理和转换是非常常见的。Python 在此领域可以大量开源媒体处理软件包协同工作。

8.10.1 二维码显示

二维码是条形码标准之一，我们现在日常使用的二维码标准是日本 Denso Wave 公司于 1994 年发明的 QR Code（Quick Response Code）。QR Code 除了常见的标准版本外，还有其他变种和分支，如下所示。

- QR Code Model 1/2，最初版本 QRC；
- Micro QR Code，单一方向 QRC，可以打印在非常有限的面积上；
- iQR Code，可以构建正方形和长方形 QRC；

- SQRC，和标准 QRC 外观一致，具备阅读限制功能，可以保存私有信息；
- Frame QR，内置画布区域，可以满足个性需求。

图 8-7 中展示了该公司推出的 Micro QR 二维码与 QR 二维码的差异。Denso Wave 拥有二维码发明专利，但是对不同分支采用不同的许可证，大规模商用前请联络该公司了解详情。

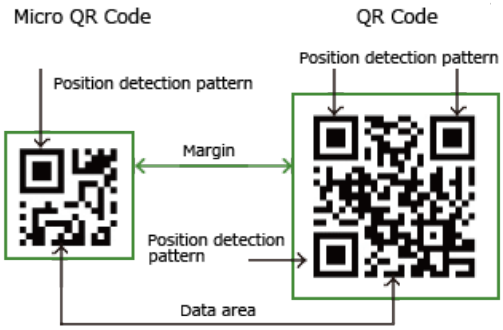


图 8-7　Micro QR 二维码与 QR 二维码对比图，来自 Denso Wave 官网

物联网的最初应用：RFID，RFID 推出的最初目的就是在各个行业中替代条形码，以实现可读/写的安全标签。当初的主流想法是，条形码是只读的方式，所以不适用于安全敏感型应用。但是不得不遗憾地说，好像这种想法被互联网企业颠覆了。在移动互联网应用中，二维码已经非常普遍。主要的应用如下：

- 文字传输（QPython 支持通过二维码代码传输脚本，例如 Python 代码）；
- 网址导航；
- 电子邮件发送；
- 电话拨打；
- 短信发送；
- 地理位置转发；
- 虚拟名片；
- Wi-Fi 密码；
- 二进制压缩文件；
- 资金转账和消费。

所有应用首先来自最基础的应用：文字传输。二维码的其他语言是利用了互联网的各种 URI 而已。根据第三方测试，智能手机支持的 URI 在 iOS、Android、Windows Phone 8 之间是类似的。其实不仅仅标准 URI 可以得到支持，甚至一些 APP 也可以开发自己定义的 URI。这和内置 HTML5 的 URI 支持有关。和 RFID 一样，二维码可以用于手机某些功能的开关。URI 通用格式如下：

```
scheme://host:port/path
```

QR 二维码支持的常见 URI 列表可参见表 8-3。

表 8-3　QR 二维码支持的常见 URI 列表

用　途	URI
网址导航	http://，https://
电子邮件	mailto:outsource@126.com?subject=Hello%20Dude&body=I%20love%20is%20Python.
电话拨打	tel:1-408-555-5555
短信发送	sms:1-408-555-1212
地理位置	geo:13.4125,103.8667（请查看 RFC5870 标准，还可以用 HTTP URL 承载地理信息）
Wi-Fi 密码	WIFI:T:WPA;S:MM;P:123456;H:true（类型：WPA；SSID：MM；密码：123456；隐藏）
Facetime	facetime:user@example.com
谷歌市场	market://details?id=org.example.com

此外还有电子名片 vCard、电子日历 vCalendar、MECARD/BIZCARD/MMS（彩信）。许多格式需要采用多行文本内容，无法在表格中罗列，这有待读者自己去发掘。不过，我们这里介绍一下 vCard 的内容格式。vCard 比其他单行信息要复杂，必须注意回车、空格和使用 UTF-8 编码：

```
BEGIN:VCARD

N:Kai,Liu

TEL;CELL;VOICE:+86-1380178XXXX</p>
EMAIL;PREF;INTERNET:allankliu@xxx.com

ORG:Ennovation LLC.,

URL:http://blog.sina.com.cn/allankliu

END:VCARD
```

接下来，可以使用 python-QRCode 将 vCard 转换成对应的 QR 图形。

我们要客观地评估 QRC 和 RFID 之间的竞争。在没有网络的情况下，RFID 在近场通信方面是优选；但是在有网络的情况下，网络安全算法可以弥补 QRC 的不足。不过，动态 QRC 在某些情况下是必需的。因为 QRC 毕竟是可以扫描的，黑客可以充分利用高清摄像头进行中等距离扫描，进行信息和资金的截获和窃取。

所以，在利用上述 URL 和信息交换技术的场景中，可以根据字符串来动态显示二维码。Python 可以很容易地处理这件事情。Python 生成 QRcode，有三个候选包：

- python-qrcode；
- pyqrcode；
- pyqrnative。

E-Badge 显示器

E-Badge 是笔者设计的基于电子墨水屏（EPD，E-Paper Display）的二维码模块，衍生自笔者研发的电子货架标签系统。其机械结构分解图如图 8-8 所示，而显示数据产生流程如下。

（1）输入字符串：输入用户字符串，如 vCard、网址等；

（2）图形生成：由 PIL 和 QRcode 软件包生成 EPD 分辨率的二进制图形文件；

（3）数据传输：上位机通过 pyserial 和定制协议将二进制图形文件传输给内置的 LPC1114 MCU；

（4）图形显示：LPC1114 将二进制文件先缓存到外部 FeRAM，然后再从 FeRAM 中将数据打印到 EPD 屏幕上。

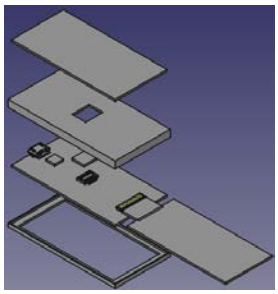


图 8-8　E-Badge 机械结构分解图

在数据传输环节中，应客户要求，增加了 AES 算法，以避免产生未经授权的访问操作。

E-Badge 作为一种通用的显示模块，其显示位图部分可以由 MCU 生成，更多场景中由主机端 Python 生成。图 8-9 展示了二维码和中文的显示效果。产生 QR 图形的代码如下：

```
import qrcode
import qrcode.image.svg

if method == 'basic':
    # Simple factory, just a set of rects.
    factory = qrcode.image.svg.SvgImage
elif method == 'fragment':
    # Fragment factory (also just a set of rects)
    factory = qrcode.image.svg.SvgFragmentImage
else:
    # Combined path factory, fixes white space that may occur when zooming
    factory = qrcode.image.svg.SvgPathImage
```

```
img = qrcode.make('Some data here', image_factory=factory)
```

图 8-9　E-Badge 模块中 EPD 显示二维码和中文的效果

python-qrcode 自带工具软件，可以在命令行中产生 QR 图形：

```
qr --factory=pymaging "Some text" > test.png
```

图 8-9 为最终显示的效果。作为一个模块，它能够做的事情很有限，即动态显示。但配合互联网 socket/urllib2 和 crypto 加密算法，其可以实现：

- 工厂现场管理；
- 商务信息推送；
- 门禁控制；
- 金融支付；
- 物流跟踪等；
- ……

扫描二维码就相当于用户单击二维码对应的网址。网址被触发后，服务器可以反向推送信息到指定的联网执行器，同时服务器还可以获得用户扫描的地理位置。如果配合其他传感器或系统，如路由器、视频监控等，还可以同时获取用户截图和扫描的设备信息。这还只是使用通用扫描工具，如果使用专用 APP 扫描，则会获取更多用户信息。因此，二维码扫描不仅仅是显示一张点阵图而已！

8.10.2　多媒体相关软件包

媒体采集、识别、处理、智能算法，配合网络通信，可以实现更多酷炫的物联网应用。

8.10.2.1　audiolazy

它是支持表达式的实时声学数字信号处理（DSP）包。Python 有许多科学计算分析包以及绘图包，也支持表达式，但大多数是在既有数据集上的计算。有了这个音频级别的 DSP 包，使得 Python 可以处理音频以及各类时序数据流（只要数据采样率在音频范围内）进行实时分析，并在此基础之上实现频谱分析和报警。

audiolazy 是纯 Python 包。但是根据应用，用户可能需要安装科学计算类、绘图类、GUI 之类的库。此外，使用 audiolazy 要求开发者对函数式编程和 DSP 算法等知识有所了解。

8.10.2.2　pyo

pyo 是使用 C 编写的 DSP 包，可以用于音频信号分析和产生音频信号，它包含了音频信号处理的许多类。

使用 pyo，用户可以将信号处理直接嵌入 Python 脚本或者工程中，并实时操纵这些信号的变化。在 pyo 工具中提供了各种操作，如音频信号的算术（叠加等）计算、基础信号处理（滤波器、延时、合成产生器等），还包括复杂算法来产生声音特效和其他有创意的音频。pyo 还支持 OSC（Open Sound Control）协议来阐述声音特效和控制流程参数，简化 MIDI 控制。pyo 还可以产生复杂的音频信号处理链。很明显，这是一个针对艺术家和音频工程师设计的模块。

若想安装 pyo，必须到官网下载对应的安装包：http://ajaxsoundstudio.com/software/pyo/。

pyo 实时产生的音频信号可以在这个网址收听：http://radiopyo.acaia.ca/。

某些音频软件将 pyo 作为音频引擎，包括软件合成器、音频处理器等。以下是一个产生旋律的简单例子：

```
from pyo import *
s = Server().boot()
s.start()
wav = SquareTable()
env = CosTable([(0,0), (100,1), (500,.3), (8191,0)])
met = Metro(.125, 12).play()
amp = TrigEnv(met, table=env, mul=.1)
pit = TrigXnoiseMidi(met, dist='loopseg', x1=20, scale=1, mrange=(48,84))
out = Osc(table=wav, freq=pit, mul=amp).out()
```

8.10.2.3　dejavu

dejavu（法语：似曾相识）。Python 的 dejavu 包是音频指纹和识别算法的一个实现。使用 dejavu 过滤一次音频，就可以产生对应的音频指纹。通过播放歌曲和录音麦克风输入，dejavu 会尝试匹配音频指纹，找到对应的歌曲。dejavu 以及音频指纹的应用场景主要是识别歌曲，保护版权；同时可以实现对于互联网内容的自动监控。此外，还可以实现内容完整性校验、辅助水印等。

8.10.2.4　python-sox

SoX（Sound Xchange）是一个跨平台的命令行工具，用于多种计算机音频格式之间的转换，同时还支持在这些音频文件上添加音效。最后，SoX 可以播放音频及录制音频。在 Linux 中，最常见的音频处理库是 libsox，其对应 Python 封装为 python-sox。以下是 python-sox 的参考

代码：

```
import pysox

#open an audio file
testwav = pysox.CSoxStream("test.wav")
#create an audio file with the same parameters as the input file
out = pysox.CSoxStream('out.wav', 'w', testwav.get_signal())

#create an effects chain using the signal and encoding parameters of our files
chain = pysox.CEffectsChain(testwav, out)
chain.add_effect(pysox.CEffect("vol",[b'18db']))
chain.flow_effects()
#cleanup
out.close()
```

8.10.2.5 其他语音软件包

此外，Python 在音频领域的其他软件包还有：

- python speech recognition，Python 的语音识别；
- pyttsx，Python 的 TTS（文本语言合成）软件包；
- PyMedia，C/C++/Python 的多媒体模块，可以对 MP3/OGG/AVI 等多媒体格式文件进行编码、解码和播放，基于 ffmpeg 提供了简单的 Python 接口。

看来，Python 在多媒体领域的能力还是很强的。

8.10.2.6 ImageHash

除了音频指纹，还有图形指纹。图形指纹的应用场景如下：

- 利用图形指纹检索不雅照片，并给出警告，过滤媒体。这是将其用于网络防范色情照片的原因。
- 建立图片搜索引擎，并记录图片及其出现的网址。
- 整理个人图片收藏，并删除重复的照片。

ImageHash 支持三种 Hash 算法：

- aHash（average Hash），均值 Hash；
- pHash（perception Hash），感知 Hash；
- dHash（difference Hash），差分 Hash。

1．均值 Hash

算法：

- 将尺寸缩小到 8×8；
- 简化色彩到 64 级灰度；
- 计算像素平均灰度值；

- 比较像素灰度；
- 计算 Hash（64 位整数）。

图片物理尺寸、纵横比不影响计算结果值；改变对比度、亮度和改变颜色，对计算结果影响也不大。计算速度超快。

2．感知 Hash

算法：

- 将尺寸缩小到 8×8；
- 简化色彩到 64 级灰度；
- 计算 32×32 点 DCT（离散余弦变换）；
- 缩小 DCT 结果，保留低频；
- 计算平均值，进一步缩小 DCT；
- 计算 Hash（64 位整数）。

改变图片高度、亮度甚至颜色，都不会改变 Hash 值，其计算速度快。

3．差分 Hash

dHash 和 aHash 类似，但 aHash 关注均值，pHash 关注频率，dHash 跟踪的是梯度变化。

算法：

- 将尺寸缩小到 9×8（不是 8×8）；
- 将颜色简化到 72 种；
- 计算相邻像素间的梯度变化；
- 定义 Hash 输出。

传统 MD5/SHA-1/SHA-256 的设计目的是为了增加 Hash 的杂散性，而 ImageHash 是为了检查图形之间的差异性。越小的差别，其 Hash 的差别应该越小。其算法等效于计算海明距离（Hamming Distance）。所谓海明距离就是长度相同的字符串在相同位置上不同字符的个数。计算图片相似度就是计算两者 Hash 的海明距离。无论是何种算法，每张照片都有自己的唯一 Hash。相似照片的 Hash 也是类似的，其相似度通过简单比较即可以实现。

ImageHash 依赖于 PIL/Pillow 及 scipy.fftpack（pHash）。

```
from PIL import Image
import imagehash
hash = imagehash.average_hash(Image.open('test_1st.png'))
print (hash)
hash2 = imagehash.average_hash(Image.open('test_2nd.png'))

print (hash2)
print (hash == hash2) #判断两个 Hash 是否一致
print (hash-hash2) #计算两个 Hash 之间的差异
```

8.10.2.7 OpenCV

物联网诸多应用中会使用到图像及视频识别技术。OpenCV 是计算机图像识别的重要平台，也有对应的 Python 绑定。其图像识别技术的应用如下。

- 文字和字符识别：卡片输入、稿件输入、文件处理、信函分拣、自动排版。
- 机器视觉识别：自动化视觉检测、装配线自动化。
- 生物医学：血细胞计数、修复手术控制。
- 工业应用：产品质量检测、集成电路设计与测试。
- 预报：天气预报、地震预报、工业烟雾预报等。
- 其他：人脸检测、车牌检测等。

简单示例如下：

```
import cv2

img = cv2.imread("C:\\opencv\\demo\test.jpg")
cv2.namedWindow("Image")
cv2.imshow("Image", img)
cv2.waitKey (0)
cv2.destroyAllWindows()
```

计算机视觉和 OpenCV 的领域是一个非常专业的领域，超出了本书范畴。

关于 Python OpenCV，推荐读者阅读：*OpenCV Computer Vision with Python*，作者为 Joseph Howse；此外还有一本 *Programming Computer Vision with Python*，作者为 Jan Erik Solem。另外，有关机器视觉与机器学习、人工智能的内容，请阅读 *Learning scikit-learn: Machine Learning in Python*，作者为 Raúl Garreta、Guillermo Moncecchi。

8.10.3 地理位置

LBS（Location Base Service）是移动互联网和物联网应用的常见应用场景。物联网中最传统的资产定位、车队管理、船舶管理应用就是 LBS 应用之一。

GeoHash

和 ImageHash 类似，GeoHash 可以利用 Hash 来获取附近的地理位置信息。确切地说，其是将经纬度信息转换为可以排序、比较和利用通配符操作的字符串，这样在数据库检索中可以实现更快的速度。

MySQL 早期版本并不支持地理位置类型，一般的做法是将经纬度信息以浮点数形式保存在两列中。检索附近位置的做法是做矩形比较计算，速度较慢。现在 MySQL 已经支持空间函数（spatial function）和计算，但是其依然有所限制。具体情况可以参考本章延伸阅读部分。

而 GeoHash 可以将地理位置比较计算转化为字符串的通配符操作，同时 GeoHash 字符串可以建立索引，其速度更快。MySQL 的字符串通配符操作采用“WHERE LIKE”语法进行检索。

```
>>> import Geohash
>>> print 'Geohash for 42.6, -5.6:', Geohash.encode(42.6, -5.6)
Geohash for 42.6, -5.6: ezs42e44yx96
>>> print 'Geohash for 42.6, -5.6:', Geohash.encode(42.6, -5.6, precision=5)
Geohash for 42.6, -5.6: ezs42
>>> print 'Coordinate for Geohash ezs42:', Geohash.decode('ezs42')
Coordinate for Geohash ezs42: ('42.6', '-5.6')
>>> print 'Exact coordinate for Geohash ezs42:\n', Geohash.decode_exactly('ezs42')
Exact coordinate for Geohash ezs42:
(42.60498046875, -5.60302734375, 0.02197265625, 0.02197265625)
```

8.11　通信报文分析

物联网最大的任务是实现联网通信，针对各层通信报文进行分析是其日常任务之一。一些常见的网络通信，如 DHCP、DNS、HTTP 以及无线协议，对于底层协议开发者来说非常重要。通常，开发者使用 Wireshark/Ethereal、tcpdump、pcap 等工具进行报文分析。除了工程人员外，这些工具的最大用户群是对于网络安全特别感兴趣的安全专家和黑客。通信报文需要采集、过滤、着色和分析。尤其在生产环境中，报文分析是一个繁重的工作。Wireshark 内置插件采用 Lua 编写。

报文捕获，经常保存为文本格式或者某种私有二进制数据。由于报文捕获数据属于离线数据，因此任何编程语言都可以实现数据分析目的，Python 标准库和内置类型可以非常方便地进行数据处理。此外，还有专用的 Python 数据分析模块。通常是基于 libpcap 的 Python 封装，如 pypcap/pypcap3。不过，pypcap 的维护不太积极。在 Windows 下可使用 Wireshark 自带的 winpcap。

```
>>> import pcap
>>> for ts, pkt in pcap.pcap():
...     print ts, `pkt`
...
```

捕获报文后，可以利用文本处理和 struct 对报文进行解析。这属于比较传统的任务。

8.11.1　PyShark

PyShark 是 tshark 的 Python 封装，允许 Python 使用 Wireshark 解析器 dissector 进行报文解析。PyShark 并不直接解析报文，而是利用 Wireshark 命令行工具 tshark 导出 XML 来解释报文。PyShark 支持利用 Wireshark 已安装的解析器对离线报文文件和实时报文进行解析。

报文文件解析：

```
>>> import pyshark
>>> cap = pyshark.FileCapture('/tmp/mycapture.cap')
>>> cap
<FileCapture /tmp/mycapture.cap (589 packets)>
>>> print cap[0]
    Packet (Length: 698)
    Layer ETH:
            Destination: BLANKED
            Source: BLANKED
            Type: IP (0x0800)
    Layer IP:
            Version: 4
            Header Length: 20 bytes
            Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT
(Not ECN-Capable Transport))
            Total Length: 684s
            Identification: 0x254f (9551)
            Flags: 0x00
            Fragment offset: 0
            Time to live: 1
            Protocol: UDP (17)
            Header checksum: 0xe148 [correct]
            Source: BLANKED
            Destination: BLANKED
```

实时报文解析：

```
>>> import pyshark
>>> capture = pyshark.LiveCapture(interface='eth0')
>>> capture.sniff(timeout=50)
>>> capture
    <LiveCapture (5 packets)>
>>> capture[3]
    <UDP/HTTP Packet>

    for packet in capture.sniff_continuously(packet_count=5):
        print 'Just arrived:', packet
```

Brian Warner 写的博文中介绍了关于如何利用 PyShark 来捕获 RTP 协议传输的音频文件，并将传来的 Raw Audio 利用 SOX 转换成 WAV/MP3 格式。欲了解相关内容，请查看本章延伸阅读部分。

要能够解析各种无线网络协议，需要开发者研发专门的解析器。在一些 BLE、6LowPAN 和 Zigbee 方案供应商提供的资料中可以找到对应的解析器。笔者也推荐依据 Wireshark 展开各类物联网协议分析。

8.11.2　pypcapfile

pycapfile 是 libpcap 捕获数据文件的解析器，作者为 Kyle Isom。

```
>>> from pcapfile import savefile
>>> testcap = open('test.pcap', 'rb')
>>> capfile = savefile.load_savefile(testcap, verbose=True)
[+] attempting to load test.pcap
[+] found valid header
[+] loaded 11 packets
[+] finished loading savefile.
>>> print capfile
little-endian capture file version 2.4
microsecond time resolution
snapshot length: 65535
linklayer type: LINKTYPE_ETHERNET
number of packets: 11
```

8.11.3　scapy 和 scapy3k

scapy 和 Python 3 版本 scapy3k，是一个完整的 TCP/IP 网络报文嗅探和解析工具包，作者为 Eriks Dobelis。

scapy 是一款强大的交互报文操纵程序，它可以伪造和解码多种协议报文，发送或捕捉报文，匹配请求并回复报文等。该软件可以处理经典任务，如扫描、Traceroute、嗅探、单元测试、攻击和网络发现等；还可以执行某些特殊任务，如发送无效帧、插入 802.11 帧、整合性黑客技巧。总的来说，这是一款针对以太网和 Wi-Fi 攻防的工具，适合安全专家和黑客使用。

8.11.4　pcap Web 分析

在 GitHub 上有一个项目：Pcap Analyzer Online。其设计是在本地运行一个 Web Server，从浏览器上传 pcap 数据，由 Server 对数据进行分析后以图文形式展开报告。该代码没有针对流式数据进行整合，仅适合小规模数据；其缺乏安全机制，也不适合投入生产环境。但它的确可以作为后台运维管理的代码基础。

其代码依赖于 Flask、SemansticUI、PyShark、Chartkick、jQuery、Highcharts 等。类似的设计还有 Cloud-Pcap、ForensicPcap。

8.12　与 Arduino/mbed 相关的 Python 包

作为开源硬件的领军品牌，Arduino 和多种语言积累了大量的通信协议和程序。在 Arduino

Playground 栏目中有单独的 Python 相关应用的介绍。

8.12.1 Arduino Prototyping

Python Arduino Prototyping API V2 可以让开发者利用 Python 的快速原型开发能力来构建 Arduino 程序。其主要设计思路是利用串口构建的点对点协议，将硬件资源暴露给上位机来控制。而上位机 Python 代码通过该协议来控制 Arduino 硬件。

（1）在 Arduino 中下载运行 prototype.pde；

（2）在 PC 上运行 sample.py。sample.py 会包含 Arduino.py 模块，是不是类似于 SL4A 的外观模式？

prototype.pde：

```
#ifndef SERIAL_RATE
#define SERIAL_RATE         115200
#endif

#ifndef SERIAL_TIMEOUT
#define SERIAL_TIMEOUT      5
#endif

void setup() {
    Serial.begin(SERIAL_RATE);
    Serial.setTimeout(SERIAL_TIMEOUT);

    int cmd = readData();
    for (int i = 0; i < cmd; i++) {
        pinMode(readData(), OUTPUT);
    }
}

void loop() {
    switch (readData()) {
        case 0 :
            //set digital low
            digitalWrite(readData(), LOW); break;
        case 1 :
            //set digital high
            digitalWrite(readData(), HIGH); break;
        case 2 :
            //get digital value
            Serial.println(digitalRead(readData())); break;
        case 3 :
            // set analog value
            analogWrite(readData(), readData()); break;
```

```
        case 4 :
            //read analog value
            Serial.println(analogRead(readData())); break;
        case 99:
            //just dummy to cancel the current read, needed to prevent lock
            //when the PC side dropped the "w" that we sent
            break;
    }
}

char readData() {
    Serial.println("w");
    while(1) {
        if(Serial.available() > 0) {
            return Serial.parseInt();
        }
    }
}
```

arduino.py：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import serial

class Arduino(object):

    __OUTPUT_PINS = -1

    def __init__(self, port, baudrate=115200):
        self.serial = serial.Serial(port, baudrate)
        self.serial.write(b'99')

    def __str__(self):
        return "Arduino is on port %s at %d baudrate" %(self.serial.port, \
            self.serial.baudrate)

    def output(self, pinArray):
        self.__sendData(len(pinArray))

        if(isinstance(pinArray, list) or isinstance(pinArray, tuple)):
            self.__OUTPUT_PINS = pinArray
            for each_pin in pinArray:
                self.__sendData(each_pin)
        return True

    def setLow(self, pin):
```

```
        self.__sendData('0')
        self.__sendData(pin)
        return True

    def setHigh(self, pin):
        self.__sendData('1')
        self.__sendData(pin)
        return True

    def getState(self, pin):
        self.__sendData('2')
        self.__sendData(pin)
        return self.__formatPinState(self.__getData()[0])

    def analogWrite(self, pin, value):
        self.__sendData('3')
        self.__sendData(pin)
        self.__sendData(value)
        return True

    def analogRead(self, pin):
        self.__sendData('4')
        self.__sendData(pin)
        return self.__getData()

    def turnOff(self):
        for each_pin in self.__OUTPUT_PINS:
            self.setLow(each_pin)
        return True

    def __sendData(self, serial_data):
        while(self.__getData()[0] != "w"):
            pass
        serial_data = str(serial_data).encode('utf-8')
        self.serial.write(serial_data)

    def __getData(self):
        input_string = self.serial.readline()
        input_string = input_string.decode('utf-8')
        return input_string.rstrip('\n')

    def __formatPinState(self, pinValue):
        if pinValue == '1':
            return True
        else:
            return False

    def close(self):
```

```
        self.serial.close()
        return True
```

sample.py：

```
from arduino import Arduino
import time

b = Arduino('/dev/ttyUSB0')
pin = 9

#declare output pins as a list/tuple
b.output([pin])

for xrange(10):
    b.setHigh(pin)
    time.sleep(1)
    print b.getState(pin)
    b.setLow(pin)
    print b.getState(pin)
    time.sleep(1)

b.close()
```

在以上设计中，Python 主机与 Arduino 之间的通信协议采用了最简单的二进制协议。在实际工程使用中，读者可以根据自己的需要进行修改。

8.12.2　pyFirmata

Firmata 协议是 7 位串口协议，其最初的协议来自 MIDI 串口控制协议。Firmata 在 Arduino 应用中比较多见，可以将 Arduino 通过 Firmata 暴露给上位机，从而实现上位机直接控制 Arduino 硬件资源的目的。pyFirmata 是 Firmata 的 Python 封装，方便 Python 开发者访问 Arduino 硬件。笔者曾经研究过 Firmata，不太推荐开发者在工程设计中采用 Firmata。虽然有许多语言支持 Firmata，但受到历史限制，Firmata 不适合作为一个通用的通信协议。其缺乏通用性和扩展性。

8.12.3　Py2B

Py2B 是 ASCII 串口协议。和 Firmata 采用 7 位传输数据不同，Py2B 采用两个 ASCII 字符串来传输对象和数值：将 port11 数值设为 65，对应的传输字符串为 0x0B 和 0x41。这种方式也有限制：ADC 限制在 8 位精度，而且是有符号整数。

8.12.4　CmdMessager

CmdMessager 是基于 ASCII 的串口协议。格式如下：

```
Cmd Id, param 1, [...] , param N;
```

其中“,”作为参数分隔符，“;”作为命令分隔符。CmdMessager 支持 C#/Mono/Python 语言。pycmdmsgdemo.ino：

```
/* ----------------------------------------------------------------------------
 * Example .ino file for arduino, compiled with CmdMessenger.h and
 * CmdMessenger.cpp in the sketch directory.
 *----------------------------------------------------------------------------*/

#include "CmdMessenger.h"

/* Define available CmdMessenger commands */
enum {
    who_are_you,
    my_name_is,
    sum_two_ints,
    sum_is,
    error,
};

/* Initialize CmdMessenger -- this should match PyCmdMessenger instance */
const int BAUD_RATE = 9600;
CmdMessenger c = CmdMessenger(Serial,',',';','/');

/* Create callback functions to deal with incoming messages */

/* callback */
void on_who_are_you(void){
    c.sendCmd(my_name_is,"Bob");
}

/* callback */
void on_sum_two_ints(void){

    /* Grab two integers */
    int value1 = c.readBinArg<int>();
    int value2 = c.readBinArg<int>();

    /* Send result back */
    c.sendCmdBin(sum_is,value1 + value2);

}

/* callback */
void on_unknown_command(void){
    c.sendCmd(error,"Command without callback.");
}
```

```
/* Attach callbacks for CmdMessenger commands */
void attach_callbacks(void) {

    c.attach(who_are_you,on_who_are_you);
    c.attach(sum_two_ints,on_sum_two_ints);
    c.attach(on_unknown_command);
}

void setup() {
    Serial.begin(BAUD_RATE);
    attach_callbacks();
}

void loop() {
    c.feedinSerialData();
}
```

请读者留意以上代码中回调函数的用法。

pycmdmsgdemo.py：

```
# ------------------------------------------------------------------------------
# Python program using the library to interface with the arduino sketch above.
# ------------------------------------------------------------------------------

import PyCmdMessenger

# Initialize an ArduinoBoard instance.  This is where you specify baud rate and
# serial timeout.  If you are using a non ATmega328 board, you might also need
# to set the data sizes (bytes for integers, longs, floats, and doubles).
arduino = PyCmdMessenger.ArduinoBoard("/dev/ttyACM0",baud_rate=9600)

# List of command names (and formats for their associated arguments). These must
# be in the same order as in the sketch.
commands = [["who_are_you",""],
            ["my_name_is","s"],
            ["sum_two_ints","ii"],
            ["sum_is","i"],
            ["error","s"]]

# Initialize the messenger
c = PyCmdMessenger.CmdMessenger(arduino,commands)

# Send
c.send("who_are_you")
# Receive. Should give ["my_name_is",["Bob"],TIME_RECIEVED]
msg = c.receive()
print(msg)
```

```
# Send with multiple parameters
c.send("sum_two_ints",4,1)
msg = c.receive()

# should give ["sum_is",[5],TIME_RECEIVED]
print(msg)
```

以上各种方案都是在 USB/UART 上建立的“通用”协议。与上面的各个通信协议相比，笔者想推荐读者使用其他更加标准化的数据交换协议。

- 二进制协议：开源项目 AVRLib 中的 STX/ETX 协议、Google protobuf、msgpack 协议；
- 文本协议：JSON/mbed RPC。

上面推荐的二进制协议更加标准化，易于实现数据对象串行化。如果少量信息交换，可以使用文本协议以减少二进制报文处理的任务。在点对点的通信中，ASCII 编码并没有增加多少系统消耗。对于性能有要求的数据采集应用，如果通信带宽利用率要求比较高，可以采用 STX/ETX 协议。

在 Arduino Playground Python 部分中，还有其余的有趣工程，读者可以去浏览。

- arduino_serial.py，使用 Python 3 的 termios 和 fcntl 访问 Linux 串口。
- 使用 Python 将 Arduino 变身为 IRC 机器人。
- BSD UNIX 系统的 Arduion 串行工具。
- Python-Arduino 实时绘图，会在本章虚拟仪器部分介绍。

8.12.5 mbed

相比于 Arduino 的旧时代烙印，mbed 是面向物联网优化的平台，其提供从芯片 API、RTOS、安全中间件、各类物联网协议栈到生产配置工具等一站式解决方案。

8.12.6 mbed RPC

ARM mbed 官方推荐 RPC（Remote Procedure Call）协议通过串口和 HTTP 端口进行多机进程间通信。mbed 同时针对 MATLAB、LabVIEW、Python、Java 和.NET 提供了 RPC 库。

RPC 的格式如下：

```
/<object>/<method><params, delimited by space>

/DigitalOut/new LED1 myled
/myled/write 1
/RangerFinder/run
/Ranger/read
```

mbed 定义的 RPC 格式采用“/”来隔断对象、命令和参数，而参数采用空格进行隔断。如果只发送“/”，则可以一次返回所有可供操作的对象列表。

从 RPC 规格定义上可以看出 REST API 对该格式的影响。但是 REST API 的参数分割会采用“?”和“&”进行隔断，或者继续采用“/”分割参数，而非采用空格来隔断参数。笔者推测 mbed 这样定义与 C 语言判断边界的方式有关。其实 mbed RPC 还有改进的余地，读者可以自定义 RPC 协议。比如：

```
/<object>/<method>/params
/<objects>

object: DIx, DOx, ADx, PWMx, FNx...
method: CR, RM, UP, DL (CRUD), or read/write/toggle/run/stop
params: param1/param2/param3 ...
```

由于 MCU 的接收缓存区是有限的，太长的报文容易溢出，因此 RPC 传输内容通常为 256B，最多为 512B。

通过 mbed RPC 协议，mbed MCU 将固件中需要暴露的对象、变量和函数提供给主机，让主机可以控制从控 mbed MCU。这种思路和 Arduino 的 Prototype、Firmata、Py2B 协议非常类似，但更高明。由于 RPC 协议实际上是应用层协议，独立于传输层，因此可以在 UART/USB/Wi-Fi/BLE 上传输。

1．RPC over HTTP

```
>>> from mbedRPC import *
>>> mbed = HTTPRPC("192.168.0.4")#IP address assigned to MBED
>>> x = DigitalOut(mbed, "led1") #pass in the name of that object
>>> x.write(1)
>>> x.read()
1
>>>
```

2．RPC over Serial

```
>>> from mbedRPC import *
>>> serdev = 15 # or '/dev/tty.usbmodem1912' for Mac or '/dev/ttyACM0'
>>> mbed = SerialRPC(serdev, 9600)
>>> x = DigitalOut(mbed, "LED3") #pass in the name of that object
>>> x.write(1)
>>> x.read()
1
>>>
```

8.12.7　mbed-ls

mbed-ls（ls for mbed）是用于检测连接到主机的 mbed 设备信息的软件包，可以获取 mbed

设备的相关信息：

- 挂载点（MSD 磁盘）；
- 虚拟串口（CDC）；
- mbed TargetID 和通用名称（如 K64F）。

该软件包运行于主机 CPython，支持 Windows/Linux，可以通过 PyPI 官网安装。

基于 mbed-ls，可以开发基于 mbed+Linux/Windows 的自动化系统，可以实现的功能如下：

- 自动检测接入的 mbed 设备信息；
- 将对应固件下载到 mbed 设备中；
- 运行固件对应的主机程序，实现新的功能如生产测试、个性化配置等。

在检测系统中连接 mbed MCU 的应用如下：

```
>>> import mbed_lstools
>>> mbeds = mbed_lstools.create()
>>> mbeds
<mbed_lstools.lstools_win7.MbedLsToolsWin7 instance at 0x02F542B0>
>>> mbeds.list_mbeds()
[{'platform_name': 'K64F', 'mount_point': 'E:', 'target_id': '02400203D94B0E7724B7F3
CF', 'serial_port': u'COM61'}]
>>> print mbeds
```

除了作为 Python 模块使用，mbed-ls 还支持命令行输出及 JSON 格式输出。无论是生产环节，还是固件更新升级或实现“智能”硬件应用，都可以充分利用这个软件包。由此，Python 可以参与 mbed MCU 产品全部生命周期管理。

命令行工具的使用：

```
$ mbedls
+-------------+-----+------+------------------------+
|platform_name|mount|serial|target_id               |
+-------------+-----+------+------------------------+
|KL25Z        |I:   |COM89 |02000203240881BBD9F47C43|
|NUCLEO_F302R8|E:   |COM34 |07050200623B61125D5EF72A|
```

命令行导出 JSON 文档：

```
$ mbedls --json
[
    {
        "mount_point": "E:",
        "platform_name": "NUCLEO_L152RE",
        "serial_port": "COM9",
        "target_id": "07100200860579FAB960EFD7"
```

```
    },
    {
        "mount_point": "F:",
        "platform_name": null,
        "serial_port": "COM5",
        "target_id": "A000000001"
    },
    {
        "mount_point": "G:",
        "platform_name": "NUCLEO_F302R8",
        "serial_port": "COM34",
        "target_id": "07050200623B61125D5EF72A"
    },
    {
        "mount_point": "H:",
        "platform_name": "LPC1768",
        "serial_port": "COM77",
        "target_id": "10100000000000000000002F7F18695"
    },
    {
        "mount_point": "I:",
        "platform_name": "KL25Z",
        "serial_port": "COM89",
        "target_id": "02000203240881BBD9F47C43"
    }
]
```


mbed-ls 的硬件基础是 mbed 的 SWD debug interface 调试接口。除了 ST 的 ST-LINK debugger，大部分 mbed debug interface 曾被称为 CMSIS-DAP，现名为 DAPLink，都是开源的。要充分利用 mbed-ls，硬件上需要两枚 MCU：目标 MCU 和调试接口 MCU。变通方式是保留 SWD 插槽，代价是需要人工插入插槽。另外一种方案是，让 mbed-ls 仅参与 mbed MCU 的生产环节，而远程固件更新则采用其他 FOTA/ISP/IAP 方式。

笔者计划在下一步 mbed MCU 有关的设计中，在生产线上使用 mbed-ls，以实现固件下载和入库管理自动化脚本。

8.12.8　Python-mbedtls

mbed TLS 在被 ARM 收购之前被称为 PolarTLS；现在是开源的 TLS 实现，适合嵌入式系统。而 Python-mbedtls 是 mbed TLS 的 Python 3 封装，但该软件包是 mbed TLS 在主机端的封装，其虽然可以用于树莓派等单板机，但却不支持 mbed MCU 编程。

理论上，只要嵌入式 MCU 支持 Python 3 语法，就可以使用 mbed TLS。但实际情况是：

- mbed MCU 上运行的 Python VM 当前只有 PyMite，仅支持 Python 2。

- MicroPython VM 还依赖于 STM32/CC3100/ESP8266 的自有 API，虽然可以整合 mbed TLS，但是 C API 和标准 CPython 有区别。
- 树莓派和 Ubuntu Core 有自己的 TLS 实现。

所以，Python-mbedtls 目前仅作为开源 TLS，用于系统定制。

8.12.9 Python-xbee

XBee 是 Digi 公司出品的 Zigbee 模块，通过 USB/UART 可以与主机或 MCU 通信。xbee-api 就是 C++库，也有对应的 Java 和 Python 库。所以无论是 PC、Linux SBC 还是 MCU，都可以充分利用这个库开发 Zigbee 应用。

XBee 的串口通信协议是经典的二进制通信协议格式，对于一般嵌入式系统的通信协议有借鉴意义。

- Start：1B，0x7E；
- Length：2B；
- Frame：包括 cmdId、cmdData；
- Checksum：1B。

让我们来看看对应的 Python 语言代码：

```
#! /usr/bin/python

# Import and init an XBee device
from xbee import XBee,ZigBee
import serial

ser = serial.Serial('/dev/ttyUSB0', 9600)

# Use an XBee 802.15.4 device
# To use with an XBee ZigBee device, replace with:
#xbee = ZigBee(ser)
xbee = XBee(ser)

# Set remote DIO pin 2 to low (mode 4)
xbee.remote_at(
    dest_addr='\x56\x78',
    command='D2',
    parameter='\x04')

xbee.remote_at(
    dest_addr='\x56\x78',
    command='WR')
```

在 Linux 上采用串口访问 Zigbee 模块是一种简单直接的方式，将来采用 socket API 是主流设计。

8.13　虚拟仪器

在嵌入式系统的日常开发中，除了焊接设备、万用表，还需要用到一些专业设备：

- 示波器；
- 信号发生器（音频信号发生器）；
- 频谱分析仪；
- 数据记录仪；
- 逻辑分析仪；
- 功率计；
- 计数器；
- 网络分析仪；
- 扫频仪。

这些设备往往比较昂贵。利用 PC 加上外部高速接口是虚拟仪器的主要架构。一些高速、大缓存的仪器仪表往往使用 FPGA+FIFO+ADC/DAC 的通用架构。而中端或者入门级仪器仪表可以使用 MPU/MCU 配合外部高速 ADC/DAC/RAM 来设计。这些设备的数据采样速率和采样深度受到 ADC 采样率、RAM 容量、CPU 架构和 USB 总线速度限制。

EmbeddedArtists 的 Labtools 就是一款开源的虚拟仪器，基于 NXP LPC4370 三核 MCU：1 枚 Cortex-M4，2 枚 Cortex-M0，80Msps ADC，还单独外接一枚 LPC812。其 PCBA 如图 8-10 所示。热心开发者还提供了 Labtools 的开源 3D 模型，可以构成一台完整的仪器。

部分低端虚拟仪器的功能可以使用一些简单硬件来实现，如 Arduino、mbed 板或者树莓派就可以实现这一目的。笔者设计的一款 LPC812 开发板，就经常用于仪器仪表目的。严格地说，LPC824 或者 STM32 系列更加适合这个目的，因为这两款 MCU 都内置了 ADC。

图 8-10　EmbeddedArtists 推出的 Labtools 虚拟仪器

8.13.1 实时显示波形

示波器是实时显示信号波形的仪器，并由各种信号触发显示。专业宽频示波器是很贵的，而且是必不可少的仪表。但是我们可以利用开源硬件和软件构建一些简单的实时显示波形仪表。在网络上，可以检索到不少资源。图 8-11 是其中一种设计，来自 GitHub ArduinoPlot 工程。从总体上来说，其架构是类似的：

- 数据采集利用 UART/USB，普遍采用 pyserial；
- 数据解析后，利用绘图引擎绘制图形，大多数采用 matplotlib/pylab；
- 数据分析，如 FFT 之类的往往使用 NumPy/SciPy 进行；
- 用户界面可以在 wxPython/PyGTK 等框架中选择。

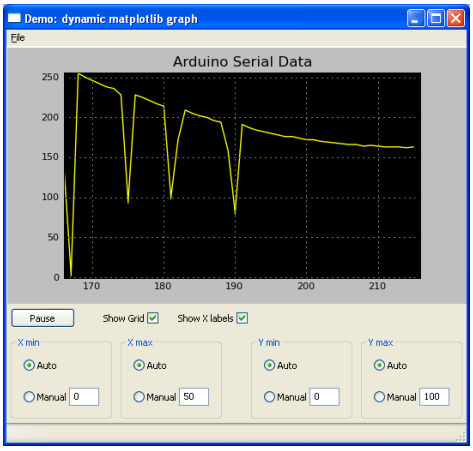


图 8-11　基于 Arduino 的虚拟示波器

由于需要实时显示波形，所以数据结构、数据通道和绘图方面需要进行优化，否则会有卡顿现象。可能需要使用以下技巧：

- pyserial 采用单独线程；
- matplotlib 采用缓冲显示；
- 数据结构采用 queue 等。

请读者自行参考本章延伸阅读部分中的相关资源和所附代码。

8.13.2 Instrumentino

虽然该软件包的名称听上去像是意大利语，但其官方下载地址却是北爱尔兰贝尔法斯特女王大学（Queen’s University）网站，作者为 Israel Joel Koenka、Jorge Sáiz、Peter C. Hauser。

该项目是一个用于 Arduino 实验仪器的模块化 Python GUI 框架。它利用 Arduino 和附加 PC

构成一个具备 GUI 的虚拟仪器。同时它还允许定义操作序列和自动化运行模式。采集的实验数据和使用记录可以自动保存在 PC 中做后道处理。Python 编写的程序也非常容易扩展。即使某些复杂设备无法使用 Arduino 进行控制，也可以使用第三方编程接口集成到 Intrumentino 框架中来。

该软件包依赖于 wxPython、pyserial、matplotlib、agw（Intrumentino）、SoftwareSerial（Controlino）。所有这一切资源都可以重用到其他更高速的 MCU 平台，如 mbed 平台。

随着开源硬件活动的普及，越来越多开发工具、仪器仪表也支持第三方插件来扩展其功能。虚拟仪器领先厂家 National Instruments LabVIEW 也提供了第三方扩展的方法，其中包括 Python 接口。

8.13.3　Vipy

Vipy（Virtual Instrument for Python）是针对虚拟仪器的许多类的集合，包括数据分析常用的 SciPy 包。但是其托管在 Google Code 上。虚拟仪器的有关代码还有：

- PyGPIBInt，Python GPIB Interface；
- Python GPIB-USB 82357A Interface。

不过，我们可以看一下托管在 GitHub 上的相关代码。

8.13.4　PyVISA

该项目可以用来控制智能测量设备，而与接口无关（GPIB、RS232、USB、Ethernet 均可）。例如，通过 GPIB 接口读取 Keithley 万用表可以使用以下代码：

```
import visa
rm = visa.ResourceManager()
rm.list_resource() # ( 'ASRL1::INSTR' , 'ASRL2::INSTR' , 'GPIB0::12::INSTR' )
inst = rm.open_resource( 'GPIB0::12::INSTR' )
print (inst.query( '*IDN' ))
```

该软件包支持 Agilent、Tektronix、Standford Research Systems 等品牌。此外，NI 单独提供了一个特殊版本的 NI-VISA。PyVISA 依赖于 NI 的 DLL 库。所以在一些小型仪表设计中，可以采用 PyUSB/pyserial/socket 替代 PyVISA。在笔者的 LoRa 项目中，在 LoRa USB Dongle 中安装了 SCPI 固件，而上位机通过 pyserial 发出 SCPI 命令，实现了 LoRa 网络的扫频、收发和报文分析目的。

笔者曾经接触过一个手机合约制造商的自动测试仪器需求，主要用于 Wi-Fi/Bluetooth/BLE 的自动测试，其需要整合 RF 衰减器，并需要通过 GPIB 连接蓝牙测试仪等专业设备，再配合一些简单的开关和指示器作为工位测试。采用此类 Python 包，则可以构建一个全自动化测试环境。

8.13.5 Pythics

Pythics（PYTHon Instrument Control System），即 Python 仪表控制系统，采用 multiprocessing 多进程设计。

Pythics 是一个 Python 应用程序包，目的是为实验室仪器和数值模拟（numerical simulation）提供简单接口。它提供了制作 GUI 的简单系统，附带必要的控件和指示器以及绘图功能。从设计角度来看，Pythics 的 GUI 和应用代码之间分割清楚，使用多线程和多进程通信，后台代码不会影响 GUI 功能。Pythics 尝试为用户隐藏所有 GUI 的复杂细节，让用户集中精力开发程序主要功能。

Pythics 的主要目的如下。

- 向用户/开发者提供一个多进程应用框架。
- 提供一个为经典科学应用而特制的简单编程方法。
- 尽可能地符合 Pythonic 哲学：尽可能使用标准库函数；尽量不引入新的特殊词汇、函数或者调用约束。

Pythics 的依赖条件：

- 需要 Python 2.6.2 以上版本，早期版本缺乏多进程支持；
- 需要 PyQt 4.5 以上版本以设计 GUI。

使用 Pythics，强烈推荐以下软件包：

- NumPy，数组支持；
- matplotlib，绘图支持。

以下库为可选项：

- PIL，Python 图形库，用于显示图形，V1.1.7 版本；
- PyQwt，专门针对科学及工程应用的 GUI widget 组件。

除了 Pythics 自身依赖项，编写完整的应用，还需要安装：

- PyVISA，VISA 实验设备的通信库；
- pyserial，用于采用 RS-232 的实验室设备通信；
- SciPy，用于附加的数值处理程序。

Pythics 由匹兹堡大学下属的纳米材料、结构与现象研究小组发布。

8.14 3D/VR/AR

3D 与 VR/AR 技术是物联网领域中的火热话题。虽然在该领域 Python 不算是优势语言，但其也有自己的应用空间。读者应该持续关注 3D 建模、无人机、人工智能、机器学习方面的交

叉应用，因为这些是未来物联网新型应用的主要方向。以下简单介绍一些 3D 领域方面的 Python 包。

8.14.1　PyOpenGL

PyOpenGL 是一个用 Python 实现的多平台 OpenGL API。PyOpenGL 支持以下特性：

- OpenGL，V1.1~V4.4；
- GLES，V~V3.1；
- GLU；
- EGL、WGL、GLX；
- GLUT、FreeGLUT；
- GLE 3（GL 挤压库）；
- GL、GLES、EGL、WGL 和 GLX 大量扩展。

PyOpenGL 包含若干个子项目：

- PyOpenGL，提供 GL、GLES1、GLES2、GLES3、GLUT、GLU、GLE、WGL、EGL 和 GLX 子包；
- OpenGL_accelerate，PyOpenGL 的 cython 加速模块，推荐安装；
- PyOpenGL-Demo，各类小型独立演示；
- OpenGLContext，基于 PyOpenGL 的教学和测试库及其他库。

其安装很简单，用 pip 命令就可以。

```
$ pip install PyOpenGL PyOpenGL_accelerate
```

PyOpenGL 需要配合许多外部库，如：

- wxPython；
- PyGame；
- PyQt/PySide；
- PyGTK；
- Raw XLib；
- OSMesa；
- Raspberry Pi BCM；
- Tkinter。

运行 PyOpenGL-demo 中的例程需要额外再安装其他软件包，最好在 Linux 系统中运行。

8.14.2 PySoy

作为 Python 的 3D 云游戏引擎，PySoy 提供面向对象的 API 用于快速游戏开发，提供渲染等关键功能，其核心层采用 C 语言开发。

所谓云游戏指的是无须下载或更新即可展开游戏的种类。基于 Python 的游戏可以运行在服务器中，显示在 Android 手机中，嵌入在网页和 XMPP 即时通信客户端或者游戏终端中。该项目推荐开发者使用 AMD GPU/APU 平台。

8.14.3 VPython

使用 VPython，即使没有太多编程经验的人，也可以很容易地创建可导航的 3D 显示和动画。本书第 10 章中还会介绍 VPython 的 3D 数据可视化在数据分析中的应用。

8.14.4 Printrun 3D 打印

物联网时代的特点之一是分布式生产、绿色能源和智能制造。3D 打印机是这一时代的典型产品。Printrun 是一个开源的 3D 打印软件包，是从开源 RepRap 3D 打印机项目中衍生的子项目。这是一个由纯 Python 编写的打印机主机软件，可以完成分层切片、零件定位、打印队列、机械控制等功能。其主要由 printcore、pronsole、pronterface 三个模块和辅助脚本构成：

- printcore.py，打印机的核心库，让编写主机应用软件更加容易；
- pronsole.py，终端控制库，支持自动补全 Tab；
- pronterface.py，pronsole 的 GUI 版本。

该项目值得大家仔细研究。首先，这个项目是一个实用项目，可以用于 3D 打印机。其次，这个软件中使用了以下几种技术，大家可以从中了解如何利用 Python 设计一个实用的物联网相关项目。

- gcode：打印机专用语言；
- pyserial：USB 串口通信；
- threading/subprocess/rpc/socket：多线程和多进程，进程间通信；
- wxPython：GUI 设计；
- pyglet：跨平台的串口和多媒体效果库；
- NumPy：科学计算，用于 3D 模型计算；
- logging：运行日志记录；
- queue：队列任务；
- decorator：装饰器。

在实际项目中观察设计者的源码并适当调试，这是学习的捷径之一。

8.15　本章小结

本章罗列了 Python 在物联网中的各种辅助开发工具，包含原型验证、虚拟仪器、定制工具、媒体处理以及一些实用项目。由于本章涉及多个方面的 Python 应用，因此读者在日常开发工作中用到的概率相当大。所以，读者应尽可能通过本章的学习来充分理解 Python 所谓的全栈开发的意义，即全方位地渗透到开发的方方面面。

第 9 章

物联网服务器端设计

服务器端技术是一个巨大的话题，这些领域的技术日新月异，迭代的速度也非常快，绝非一本书可以覆盖的。本书只能够介绍其中一部分基础的核心内容。本章主要介绍服务器相关的计算模型、接入协议、系统架构、数据存储、认证授权、UI 前端、交付与运维技术。由于数据统计、数据可视化、大数据和人工智能逐渐形成单独的技术门类，笔者将这部分数据端内容从服务器端设计中独立出来，在第 10 章中进行单独介绍。

现在咨询行业将物联网平台分为以下四大类：

- DMP（Device Management Platform），侧重设备连接管理；
- CMP（Connectivity Management Platform），侧重连接运营管理；
- AEP（Application Enable Platform），侧重应用快速开发；
- BAP（Business Analytics Platform），侧重数据分析。

平台必然是由分布式计算组成的大型系统服务，这不是本书的目标。本书的核心目的之一是为物联网设备供应商开发团队提供“入门级”的物联网设备接入服务器方案。无论是设备管理、连接管理还是应用开发、数据分析，都是按照最基本的需求进行配置的。首先让这些团队的联网设备可以拥有自己的服务器进行联网测试；继而通过弹性扩展拥有自己的云计算分布集群，以满足从小批量、中试，直至规模生产的联网需求。如果企业可以继续成长，后续超大规模设备联网，则需要持续不断地进行架构优化以满足企业的需求。架构优化是一个持续不断的过程，完成一个里程碑往往意味着新一轮代码重构，甚至推翻重新来过，这句话绝非夸张。

本书以 Python 为主，并非笔者认为 Python 在任何情况下都是最佳选择；但是 Python 的确是选择之一。Python 的许多特性使得开发者的学习曲线可以变得比较平缓。在许多云计算平台推荐的编程语言中，Python 位列其中。

在面向 Web 应用的互联网行业中，使用得比较多的语言是 Java、PHP 和 JavaScript。其中 Java 是大数据和企业级应用的主流选择，专注于企业市场的 PTC ThingWorx 和 IBM Lutos Notes 都是 Java 的典型案例。PHP 则是中小型网站的首选。在 Node.js 框架的推动下，JavaScript 在服务器端的普及速度很快。相比之下，Python 算是历史较长的一种语言。国内外著名的 Python 网站有

谷歌 GAE、新浪 SAE、FriendFeed、知乎、豆瓣、果壳、咕咚、金融界、美团、快盘、Bittorrent 等。

笔者计划在物联网相关设计中，采用 PyPy 加速、libuv 框架或 Golang 混合编程，以实现设备接入并发性能最大化。在 Java 相关的网关设备、大数据和中间件中采用 Jython 实现对接，以实现企业级应用。无论是开发周期、运行性能、应用复杂度和成熟度，都可以充分挖掘 Python 的潜力。本章内容将以 Python 在物联网各类服务器中的设计为主，兼顾一些 JavaScript 前端设计。

9.1　物联网计算模型

随着物联网云计算的标准化，云计算 PaaS/SaaS 已经逐渐成熟起来。不过从目前来看，基于私有云或公有云 IaaS 自建 IoT 服务，或混合使用 IoT Paas/Web Pass/IaaS，搭配定制服务，依然是满足物联网客户定制需求的主要方式。

9.1.1　云计算

云计算一词由谷歌推出后，风靡 IT 行业。现在云计算有三种主要的模式：

- SaaS，Software as a Service，软件即服务。
- PaaS，Platform as a Service，平台即服务。
- IaaS，Infrastructure as a Service，基础设施即服务。

这三种云计算分别代表了三种不同的抽象层次。抽象层次最高的是 SaaS，最低的是 IaaS。现在又增加了一种：Container as a Service。笔者个人觉得 CaaS 更加类似于 IaaS，是更加轻量级的虚拟机。但 CaaS 不仅仅是虚拟机，在其基础之上还带来了更多的附加收益，如多租户业务、持续集成等。究竟选择哪一种云计算模式对于物联网项目更加有利呢？从理论上来说，抽象层次越高的云计算，成熟度越高，运维需求越少，对于应用越有益。关于这方面的权衡，*Cloud Computing for Dummies* 一书介绍得很清楚。不过这个选择也不是那么简单和绝对的。因为物联网的标准碎片化和去中心化，决定了定制是其主要方向。

基于公有云，满足企业或个人某方面需求的在线软件就可以理解为 SaaS。笔者举一个更简单的例子：博客、Gmail 邮箱等都是 SaaS 服务，所以说 SaaS 历史很悠久。虽然物联网的历史也很悠久，但是有没有成熟的 SaaS 服务呢？虽然也有针对特定行业的物联网 SaaS 服务，但是好像没有普遍适用的物联网服务！这依然和物联网的碎片化和定制需求有关。虽然某些软件公司如 PTC 提供的 IoT 解决方案也很完善，但是它交付的是软件本身，而不是服务。

如果能够找到对应的 SaaS 或软件就最好了。如果没有可用的 SaaS 或者预算不足，则必然需要外包或者自主开发。配合开源软件，PaaS/IaaS/CaaS 能够为我们提供更多的技术选择。

9.1.2 Web PaaS 与 IoT PaaS

谈到 PaaS 服务，首先需要区分 Web PaaS 和 IoT PaaS。

Web PaaS 主要指的是为 Web 应用而提供的各类 Application Engine。以国外的 GAE 为鼻祖，其他的还有 SAE、BAE、JAE、ACE 等。针对 IoT 应用的 PaaS 主要指的是 Xively 类型的设备云 PaaS。

通过一番调研，笔者发现大部分 Web PaaS 并不支持物联网接入协议，主要体现在以下几点。

- 端口限制：大部分 Web PaaS 不提供 socket 接口，仅有 GAE 支持 socket 接入；
- 协议限制：大部分 Web PaaS 不支持 CoAP 协议和 MQTT 协议；
- 语言限制：大部分 Web PaaS 支持 Java/PHP/Python/Node.js 等编程语言，但都不支持 Python 的异步 I/O 框架；
- 安全限制：相当一部分 Web PaaS 无法支持单独域名下的 TLS 证书。

负载均衡、消息队列、Redis、SQL 存储、对象存储属于 Web/IoT 共享的云计算 PaaS，已经是云计算的标准服务了。

根据上述调研，基于 Web PaaS 构建物联网的限制颇多，其组成方案必须是：

- 编程语言——Java 多线程、Node.js 异步单线程，以应对高并发设备连接；
- 接入协议——仅支持 REST API 等基于 Web 的接入方式；
- 安全证书——采用明文或者供应商安全证书。

换而言之，Web PaaS 只可以构建 Web 端口的物联网应用，这对带宽、设备处理能力提出了很高的要求。所以在笔者的许多项目启动之初，都会采用 IaaS 实现。确切地说，先在 VirtualBox 虚拟机做开发，然后在云计算供应商处启动一个 Linux 实例进行部署。在 IaaS 开发中，开发环境和生产环境的差异很小。随着 IoT PaaS 的出现，情况开始出现了一点儿变化。

9.1.3 IoT PaaS 供应商

IoT 的快速发展使得许多云计算供应商在其产品线中增加了 IoT PaaS。百度、阿里、腾讯、亚马逊、微软、IBM 等都推荐将 MQTT 作为首选 IoT 接入选项；还围绕 IoT 接入增设了不少增值云服务，如大数据分析、数据可视化、授权加密等。这些都为实现基本的物联网接入提供了技术基础。现在我们来看看国内外主流云计算供应商的 IoT PaaS 产品线。

9.1.3.1 阿里云物联网套件

在图 9-1 的阿里云物联网系统框架图中，虚线内为物联网套件，主要解决了数据接入、安全认证和权限策略，并提供了 OpenAPI 和规则引擎用于应用集合。即使只有这个套件，配合手机 APP 也可以构建最简单的 IoT 应用。但是，完整的 IoT 应用还需要整合阿里云的其他服务：

- ECS，运行 Web Server；
- ACE，运行受限的 Web Server，类似于 GAE/SAE；
- OTS，表格存储，类似于 MongoDB；
- RDS，MySQL 数据库；
- DRDS，分布式 SQL 数据库；
- Lambda，大数据分析，整合离线计算和实时计算；
- ONS：消息队列，基于 RocketMQ 的 MQTT；
- OSS：开放存储服务。

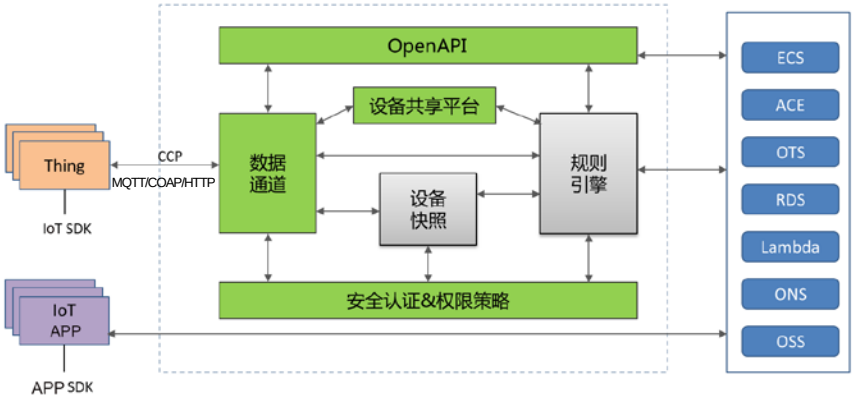


图 9-1　阿里云物联网系统框架图

注意　阿里云的产品名称经常修改，读者需要特别留意这一点。所谓物联网套件，本质上就是围绕 MQTT 而构建的一个 PaaS 服务，但是其 MQTT 数据分享双方都必须是阿里云的企业认证用户。这限制了该套件的数据分享应用范围。

读者可能注意到了，ONS 也支持 MQTT。该消息队列主要用于后端系统耦合。但是物联网套件在 MQTT 基础之上还有其他的功能整合，两者有一定的差异。

9.1.3.2　百度天工物联网

百度将其物联网云服务命名为“天工”，除了 MQTT 之类的接入服务，还提供了实时时序数据库，迎合了物联网中最常见的基于时间戳的物理量数据采集需求。其系统架构如图 9-2 所示。

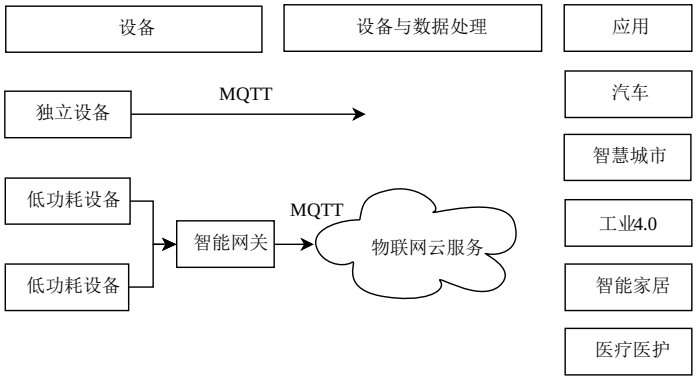


图 9-2　百度天工物联网系统架构

9.1.3.3　中国移动 OneNet

确切地说，中国移动的 OneNet 有点儿类似于 Xively。OneNet 支持以下几种接入方式。

- HTTP REST API：最常见的 REST over HTTP 方式；
- EDP：OneNet 自定义协议，一种可兼容 JSON 格式的二进制协议；
- MQTT：IBM 定义的物联网协议；
- MODBUS：施耐德定义的工业物联网协议，基于 RS232/422/485 和以太网；
- JT/T808：中国部颁标准，交通运输行业、道路运输车辆卫星定位系统终端通信协议及数据格式；
- RGMP：OneNet 自定义协议。

与其他 PaaS 相比，OneNet 面对工业、物流等行业提供了更多接入协议。但是物联网的协议太多了，需要不断地针对特定行业应用提供定制协议。

9.1.3.4　Amazon IoT

Amazon AWS 一直是云计算服务的标杆，是国内企业追赶的目标。如果网站业务面向中国境外展开，那么 Amazon 可以是首选。最近，Amazon IoT 联合许多上下游企业提供 IoT 套件。Amazon 提供 12 个月的免费账户，其中包括：

- IoT——每条信息以 512 字节计，25 万条免费消息/月，超出部分每 100 万条信息收取 5 美元。
- EC2——750 小时/月。
- S3——5GB 标准存储，20000 次获取请求，2000 次放入请求。
- RDS——750 小时/月，20GB 存储，20GB 备份，1000 万次 I/O。
- EC2 Container Registry（容器注册）——500MB 存储。

一个月最多 31 天，共计 744 小时。所以，按照时间计费的服务如 EC2，750 小时/月的计费

模式就是全月免费。AWS 的免费资源足够覆盖中小型应用在 12 个月内的使用量了。AWS 中国云计算与境外的云服务版本有一定差异，开发者使用前需要调研对比一下，并根据不同地区的法律法规做些评估。

围绕 AWS IoT，使用 MQTT 实现接入协议，可以减少带宽使用，并实现实时推送。图 9-3 展示了 AWS IoT 最简单的 MQTT 连接方案。

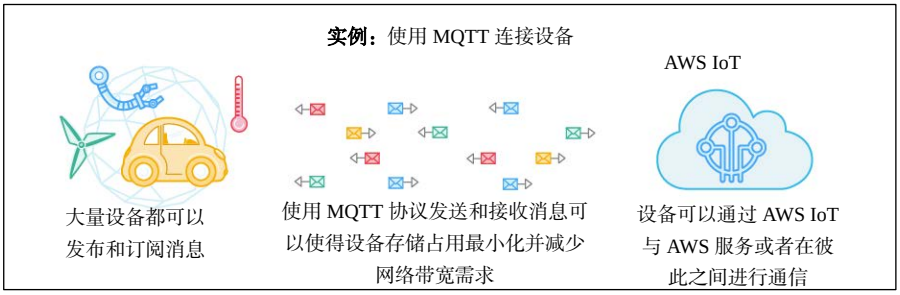


图 9-3　AWS IoT MQTT 连接方案

AWS IoT 利用 TLS 进行安全认证，构建安全通道，保护设备连接和数据。图 9-4 展示了 AWS IoT 使用 TLS 在设备和应用程序与 IoT/MQTT 服务器之间进行安全认证的流程。

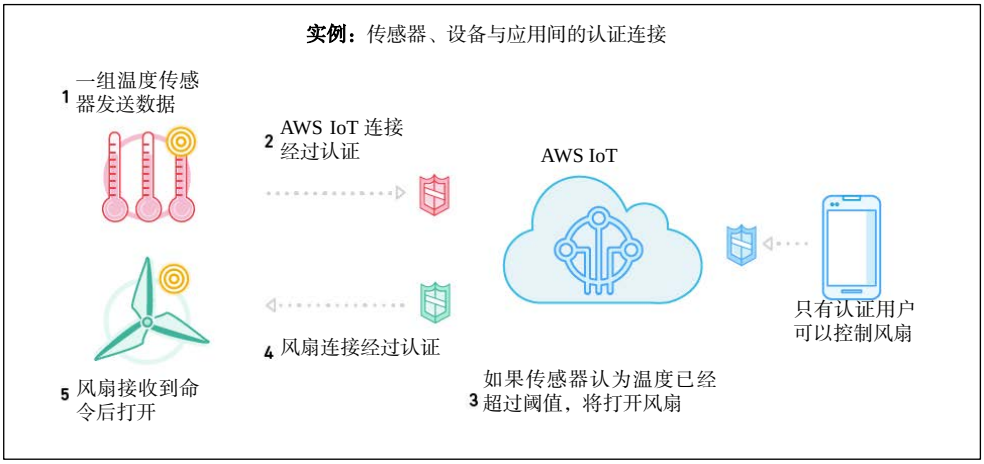


图 9-4　AWS IoT 的设备认证流程

图 9-5 是 AWS 的车联网应用场景，在云端判断车辆间的距离并实时报警，还展示了 AWS 的大数据分析、人工智能和数据可视化服务。

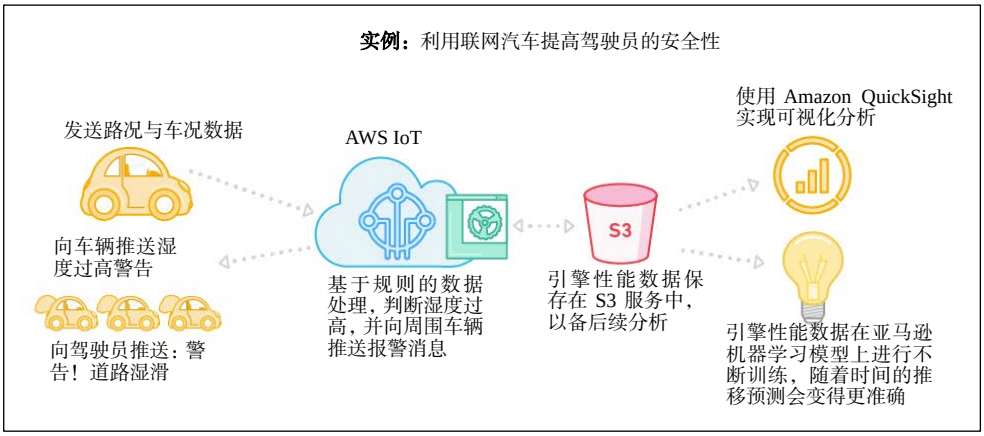


图 9-5　AWS IoT 的车联网应用场景

图 9-6 是 AWS 的专用设备联网应用场景，质谱仪间歇性地连接 AWS 并上传采集数据和状态，AWS 保存着所有最新数据。管理员可以远程获取质谱仪数据状态并设置参数。由于 MQTT 的特性，因此质谱仪与 AWS IoT 服务器之间的连接可以是间歇性的，无须维持长连接。

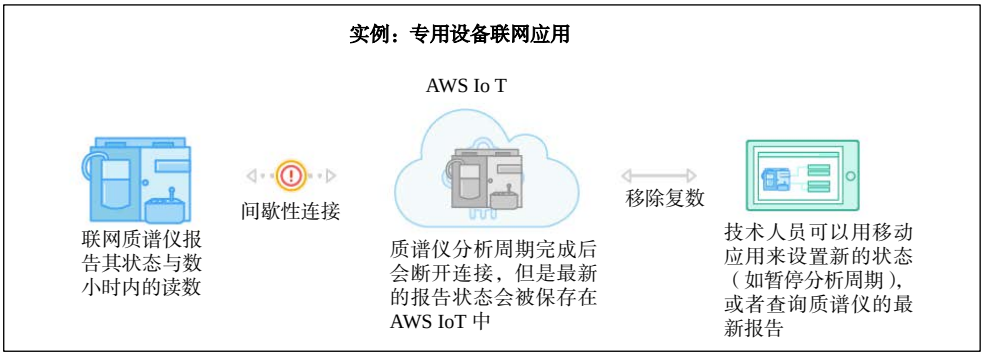


图 9-6　AWS IoT 的专用设备联网应用场景

9.1.3.5　IoT PaaS 的完整性不足

Web PaaS 配合 SQL 数据库就可以构建一个完整的 Web 应用。但是 IoT PaaS 与 Web PaaS 相比，其更像是一种负责连接的中间件。在许多 IM 供应商的宣传中，基于 MQTT 和移动 APP，就可以实现“基本”的物联网服务了。这很符合敏捷开发的原则，但完整的物联网需要更多组件：

- 数据库——数据检索、分析等；
- Web 服务器——功能配置、用户管理和 API 分享；
- 系统接入——支持 MQTT 之外的接入方式。

在实际的物联网项目中，需要单独的服务器来配合使用 IoT PaaS。所以在云计算供应商的 IoT PaaS 框图中，罗列了许多其他服务作为补充。其中包括了 ECS/EC2，也就是 IaaS 中的服务器实例。所以，真正实用的最小物联网服务器配置应该是 IoT PaaS + Web PaaS + APP。其中 IoT PaaS 负责设备对接；而 Web PaaS 运行 Web 应用，进行设备资产管理，并提供 Web 服务给 APP。

9.1.3.6　IoT PaaS 的其他顾虑

笔者常常反省：如果一个物联网项目选定某家 IoT PaaS，甚至 SaaS 平台构建物联网应用，实施周期是否更短？现在看来，依托公有云的 IoT PaaS，短期内启动工程的成功概率还是很大的。但在笔者调研 IoT PaaS 的时候，发现能够同时满足物联网在实时性、数据持久性、费用、编程语言和架构、价格、扩展、学习周期等多方面需求的方案不多。另外，由于技术不透明，许多 IoT PaaS 的潜在问题并不能够在调研阶段暴露出来。笔者个人喜欢使用开源软件，所以对闭源系统一贯抱有怀疑态度，往往脑子里会有一堆问题：

- 使用 IoT PaaS 能够满足 socket 接入、异步 I/O 框架、TLS 证书、独立域名以及 Python 扩展问题吗？
- 如果采用分布式存储方式，可以保证实时性吗？数据检索与单机 SQL 服务有区别吗？
- 如果采用消息队列 PaaS，可以保证实时性，但如何保证收发顺序呢？
- 如果选定某家 PaaS，若要切换供应商怎么办？
- 对于中小企业来说成本很重要，PaaS 的整体计价比 IaaS 低吗？
- IoT PaaS 是否能够支持非标准接入协议以支持客户的现有设备投资？

PaaS 的益处是降低开发复杂度、弹性扩展和免运维。PaaS 往往依托于大型云计算服务，一上来就可以规划较大的应用场景。无论是负载均衡、数据库还是运维，都由 PaaS 供应商保障。这一点在 Web PaaS 中体现无遗。在 IoT PaaS 上，其接入数量也会超过自建 MQTT 服务器。

不过，使用 IoT PaaS 也要付出一些代价。

1．计价方式过于复杂

阿里云的 IoT PaaS 在公测阶段免费，AWS/BlueMix/Google/Azure 都有免费额度，但是其正式商用后的长期使用价格才是我们需要了解的成本。PaaS 的价格计算很烦琐，按需计价的模式不见得对每种业务都适用，调研、计算、比较都会花去很多时间。由于 PaaS（以及 SaaS）的选择涉及技术、商务和生态，因此需要多听取有过使用经验的开发者的意见才能够做出正确选择。

2．供应商黏度大

许多中国企业在国内用阿里云，在国外用 AWS。如果不是兼容性设计的架构，则不同的 IoT PaaS 在 API 等技术细节上存在许多不同，切换供应商成本太高。

3．其他限制

出于技术、安全和商业因素，许多云计算供应商对于 IoT PaaS 有一些限定条件，比如阿里

云要求 MQTT 的分享双方都必须是阿里云的企业用户等。

综上所述，如果基于 IoT PaaS 开发物联网，则往往绕不开 IaaS。既然无法回避，那么全部基于 IaaS 开发就可能成为最终的选择。现在开源工程让基于 IaaS 的开发变得更加容易，只是需要注意服务器架构设计和技术依赖性。

基于 IaaS 的开发往往从单台单核服务器开始，之后逐步升级，小型应用系统甚至可以将数据库和代理服务器都在单机系统中运行。随着产品和市场规模的拓展，采用纵向（即提升服务器配置）和横向（即增加服务器数量）两个方向进行系统扩展，通过购买更多服务器实例和带宽，增加负载均衡、内存数据库、队列服务和 CDN 对系统进行扩展和加速，可以实现自己的物联网应用“平台”。IaaS 的缺点是开发难度大，学习门槛高，系统扩展后的服务器运维成本比较高。

9.1.4 PaaS/IaaS 混合架构

现阶段物联网云计算的设计路径可以有以下几种。

- Web 简约型：基于 Web PaaS/REST API/移动 APP 实现最简单的 Web 物联网服务。
- IoT 简约型：基于 IoT PaaS 和移动 APP 实现最简单的 MQTT 物联网服务。
- 购买型：依托 IoT 软件，如 PTC ThingWorx，在私有云和公有云服务器上安装。
- 混合型：基于 IoT PaaS 接入 MQTT；基于 Web PaaS 接入 HTTP/REST/WebSocket；基于 IaaS/Docker 支持非标准接入，利用其他云服务中间件构成物联网服务系统。
- 自建型：完全基于 IaaS 和 Docker 自建物联网服务所需要的所有组件。

工程师们认为基于 IaaS 自建对于云计算平台的依赖性会比较少。从企业的角度来看，快速、完整、成熟是关键因素。不过即便采用外包和购买软件，也需要深入了解自身需求和供应商采用的系统架构，否则后患无穷。

在云计算供应商填补了 IoT PaaS 这块拼图之后，我们可以使用 Web PaaS/IoT PaaS/IaaS 进行混合架构设计。让我们来看看混合架构是否完整。

9.1.4.1 端口与协议

现在的物联网接入方式多种多样，包括 MQTT、CoAP、WebSocket、HTTP/REST API 和客户定制协议的 socket server。我们必须采用 IoT PaaS/Web PaaS/IaaS 的混合接入模式，其对应关系如下。

- IoT PaaS：负责 MQTT 设备接入；
- Web PaaS：负责 HTTP/REST/WebSocket 设备接入；
- IaaS：负责 CoAP/LWM2M/socket 以及其他的定制协议。

此外，一个完整的物联网应用必须是可运营的，与用户、公司组织架构有关联的业务逻辑

可以通过 Web PaaS 进行部署。如果考虑与第三方整合，需要对外提供 REST API，那么也可以在 Web PaaS 中部署。这方面可以充分利用 Web PaaS 的免运维优点。

9.1.4.2　编程语言

接入协议与编程语言无关，但是 Web PaaS 对于编程语言和框架有限制。以 Python 为例，几乎所有 Web PaaS 均不支持 Python 异步框架。所以在这些 Web PaaS 中，必须使用 Python paho 来操纵 MQTT/IoT PaaS。paho 可以在大多数同步型 Python Web 框架中使用。如果基于异步框架，则可以使用 twisted-mqtt-client 库来访问 IoT PaaS。

9.1.4.3　数据持久

由于数据在 MQTT broker 处就可以转发，如果没有数据记录需求，则可以直接忽略后端服务器。但在大多数情况下，许多数据需要被记录、归纳、整理和统计，并保存在 SQL、NoSQL、NewSQL 和各类大数据存储集群中。所以，从 MQTT 到后端存储服务器之间的数据流量需要单独计算，要避免出现数据堵塞和丢失现象。IoT PaaS 与数据库中间需要单独的服务器做桥接。

9.1.4.4　安全

Web PaaS/IoT PaaS/IaaS/CaaS 都可以采用标准安全方法 SSL/TLS。一个混合云计算架构必然拥有多台服务器，对应多个外网 IP 和端口，可以采用支持子域名的 TLS 安全证书对这些端口进行安全保护。部分 IoT PaaS/Web PaaS 对域名/安全证书有一定限制。

选择 SaaS/PaaS/IaaS/CaaS 不仅仅是技术选择，还涉及商业和生态选择，变量太多。由于对于互联网和云计算缺乏足够了解，也由于本人的程序员技术背景，因此启动物联网服务器设计时，笔者并没有考虑太多，就直接在阿里云上启动了一个 Ubuntu 实例进行开发。彼时的决定固然与当时缺乏 IoT PaaS 供应商有关，但这种实用主义方式，现在看来可能缺乏足够的考虑和全局观。实际上，基于 IaaS 和 PaaS 的开发直接决定了日后的应用规模。随着物联网云计算越发成熟，笔者希望本书的读者能够在启动项目之前，仔细调研，考察清楚 IaaS/PaaS 的区别和限制，然后再开始着手实际设计。

9.1.5　雾计算

雾计算是另外一种计算模型。该术语由纽约哥伦比亚大学 Salvator J. Stolfo 教授首创，并且被 CISCO 大力推广。从名字来看，云高高在上，而雾贴近地面。云计算利用计算集群提供强大的计算能力，而雾计算则利用靠近边缘节点的各种零散的相对较弱的计算机资源进行计算。这是一种分布式、计算节点前置的模式。

在车联网场景中，红绿灯可以根据当下的车流情况进行切换。虽然云计算可以将现场采集的信息传回中心云计算节点，然后再进行红绿灯的切换，但是大量数据上传耗费通信资源，而

且通过计算并下行控制红绿灯，还会带入不可测的延时。在这些应用中，雾计算反而比云计算更加适合。

但是，笔者认为这实际上还不是雾计算的最佳案例。因为速度固然是物联网的瓶颈，而流量成本和商业模式才是雾计算的动力。雾计算需求将来自各个行业应用场景中近场范围设备数据整合的需求。

举个实例，在联网医疗设备中，原先每种设备基本都自成体系，都参考了端、管、云三层计算模型。心脏监护、呼吸机、血氧监护、药品滴注等设备都联网连接各自的后台服务器，这造成了许多数据孤岛。即使各个服务器能够在云端彼此相连，但是许多分析需要长时间分享数据，高并发、大规模、长时间的数据交换造成的带宽成本，无论对医疗器械供应商还是医疗机构都是一笔很大的费用。最终这笔费用将由患者和纳税人买单。在这种情况下，将处理节点前置到网络边缘是一件很自然的事情。

在边缘计算机中既可以采用容器方式将各家设备的算法整合运行在同一台计算机中，比如在运行于某一中心设备（如监护仪或者单独的嵌入式 Linux 计算机）中，也可以实现数据汇集和转发。数据可以在边缘节点得到实时处理；同时又以压缩方式或者流式数据方式转发给各台设备的后台服务器集群，或者保存在本地存储器中。更加重要的是，各家设备的算法依然可以是保密的，知识产权是受到保护的。所以，从流量、成本、反馈、商业生态各个方面来看，雾计算即边缘计算是对于云计算模式的一种必要补充。因此，边缘节点采用多核处理器用于运行多个不同应用进程也是很有必要的。好在现在消费级处理器都出八核、十核了，其功能够强，成本够低。笔者个人认为 ARM/MIPS 的服务器战略应该基于雾计算，而非超算和云计算。

无论是何种计算模型，都必须根据应用本身的特点（如数据流向、流量、频率等要素）进行规划；否则，容易浪费宝贵的开发时间和资金。

9.2 物联网与互联网设计异同

由于物联网的系统组成、接入标准、网络拓扑、数据特性与互联网有所不同，因此原有的互联网架构需要升级改造之后才能适应物联网的需求。如果其针对某些细分行业的共同特性和需求推出 SaaS 服务或软件，或许可以占领一部分细分市场。互联网最后势必会与物联网融合。下面我们来了解一下物联网与互联网设计之间的联系和区别。

9.2.1 基础架构

互联网和物联网可以共享许多基础架构，物联网的许多应用基于互联网基础之上，包括基于 REST API 的数据共享，基于 Web 的用户管理和轻应用，相关的负载均衡、消息队列、结构

化信息数据存储、文件存储等，都可以共享。正是互联网企业推出云计算后，更多传统企业才有机会获得更低价格的服务来构建物联网应用。

9.2.2　标准化程度

在互联网应用中，用户都通过浏览器接入，标准化程度高。简单的 Web 设计入门门槛也较低，LAMP 组合可以快速构建一些 Web 站点。如果充分利用 Web PaaS，还可以实现弹性扩展和免运维。更多的供应商以 SaaS 方式提供各种软件服务。互联网行业的迭代速度惊人，用户规模也常常以指数级上升，这成就了一些“独角兽”企业。

与互联网和移动互联网相比，物联网应用还有许多独特的需求：比如持续的高并发 TCP 长连接、UDP 通信、实时性、私有通信协议、截然相反的数据流向、快速更新频率等，以及复杂的管理架构、与 CRM/ERP 的整合等。也就是说，物联网目前还是一个需要大量定制服务的碎片化领域，无法像互联网那样通过标准化服务实现用户的指数级上升，且制约了设备销量和用户数的上涨。

9.2.3　业务模式

与互联网等轻资产行业相比，物联网因为涉及设备投入，会有固定资产投入和库存管理，这决定了物联网有别于互联网，其是重资产行业。比如传统制造业领域，物联网将成为其产品或服务的一部分，而互联网是其销售和客服的渠道。物联网设计迭代主要发生在设备端和服务器端两端，所以服务器的设计有时候不得不向设备端妥协。

9.2.4　系统构成

物联网端到端的典型系统构成如下：边缘设备、网关、服务器和 APP 客户端。首先，我们需要解释一下“网关”和“边缘服务器”这两个术语。

9.2.4.1　网关

网关（gateway），又被称为网间连接器、协议转换器。网关在网络层以上实现网络互联，其是较复杂的网络设备，仅用于两个高层协议不同的网络互联。网关既可以用于广域网互联，也可以用于局域网互联。 网关是一种充当（协议）转换重任的计算机系统或设备，用在不同的通信协议、数据格式或语言，甚至体系结构完全不同的两种系统之间。网关是一个翻译器，与网桥只是简单地传递信息不同，网关对收到的信息要解包后重新打包，以适应目的系统的需求。

9.2.4.2　边缘服务器

Edge Server。随着互联网及其应用的快速发展，客户对网站系统访问的响应时间、网站内

容以及所提供服务的可靠性、即时性等要求也越来越高，使得单台服务器支撑的网站系统已无法满足客户需求。取而代之的是采用两到三层架构的一组服务器。第一层是跟用户直接发生联系的前端服务器，也被称为边缘服务器（Edge Server）。从这个角度来说，CDN、负载均衡都是边缘服务器。

物联网的接入方式存在多种形式，所以物联网的系统构成与网络拓扑有密切关联。其大致有三种拓扑形式：

- 边缘节点（设备）、网关、（云）服务器（集群）；
- 边缘节点（设备）、边缘服务器，云服务器；
- 边缘节点（设备）、（云）服务器（集群）。

第一种拓扑与第二种拓扑存在一些差异。在大多数场景中，网关负责协议的转换，并传输给服务器，这是第一种拓扑的情况。在某些场景中，要求没有云服务器的情况下，整个网络依然能够正常工作。这个时候，在中间环节的计算节点中，边缘服务器的角色更加重要，其分担了部分服务器的计算任务。这就是第二种拓扑的情况。网关和边缘服务器可以运行在同一台计算机内，以不同的进程形式出现。尤其是一些 WSN 网络，中间节点需要同时完成协议转换（网关）和边缘节点控制计算（边缘服务器）。一些离散性的物联网应用，如共享单车的应用，采用蜂窝数据模块与服务器直接通信。此类应用无须设备组网，仅须设备联网，无须边缘服务器或网关。其适用于第三种拓扑。

传统互联网采用的是 B/S 或 C/S 模式，浏览器或者客户端与服务器采用 TCP/IP 进行通信。即使中间有代理服务器和负载均衡，也大多数是端口转发，很少做协议转换和边缘计算。不同的网络拓扑导致物联网和互联网之间，甚至不同的物联网应用系统在建模之初就存在较大的差异。IBM 的 LoRaWAN 网络交换机的模式就颠覆了传统的设计方式，可以预测这方面的创新还会不断出现。

9.2.5 设备接入协议

物联网与互联网的最大区别之一就是接入协议的多样性。当然，物联网可以借用互联网的许多现有协议，如：

- REST API over HTTP/HTTPS；
- XMPP over HTTP/HTTPS；
- RTSP/RTP/RTCP；
- WebSocket。

为了让更多嵌入式设备组网、联网，在物联网应用中更多采用轻量级协议：

- MQTT；

- CoAP；
- TCP/UDP socket；
- MODBUS 及其他行业的标准通信协议。

经过十多年的推广，IBM 的 MQTT 日渐成为主要的物联网协议。但是设备端 MCU 能力不足，不能够支持完整的 TCP/IP/TLS 协议，所以有时候不得不退而求其次，采用 CoAP 或 TCP/UDP 套接字服务器以适应设备的计算能力。

9.2.6　数据特性

除了设备接入协议不同外，物联网和互联网的差异还体现在服务器架构、存储子系统、访问控制策略、分析平台等多个方面。这些差异点的根本原因在于两类系统中不同的数据特性，包括带宽、总量、种类、方向、更新频率等。联网应用的本质是数据的流传，数据特性的不同导致了连接方式、系统架构、存储、分析、可视化等一系列差异。

互联网以人为服务对象，主要是解决人与人之间的信息交换。

在 Web 1.0 时代，内容为重点。以网站提供内容为主，新闻、视频、论坛、电子商务、电子邮件等均如此。网站流出数据大于流入数据，主要从网站流向用户。

在 Web 2.0 时代，社交是重点，内容多由用户产生。用户围绕云服务器进行数据交换，包括社交网站、微博、微信、视频和照片交换服务，服务器流入/流出数据大体相等。

物联网以设备为对象，其数据主要是设备采集的物理量，主要功能如下。

- 数据采集：物理量和使用行为等各方面数据；
- 数据分析：原始数据清洗、分析、提炼、挖掘；
- 远程控制：设备固件升级、限制、激活；
- 业务融合：与第三方平台进行融合。

我们可以发现：物联网的数据流向是从设备流向服务器，而且大多数属于高频、短小数据；另外，设备间、设备与终端之间的数据流转也非常多。现将 Web 与物联网应用的数据特点比较汇总于表 9-1 中。

表 9-1　Web/IoT 数据特点

对比项	Web	IoT
流入带宽	少	多
流出带宽	多	少
并发连接	不定	多
连接时间	短	长
峰谷时段	有规律，也有突发情况	有规律

续表

对比项	Web	IoT
总量	大	大
种类	HTML、文本、程序、多媒体	物理量、程序、媒体
速度	快	快
连接特性	短连接居多	长连接居多
传输层	TCP/UDP	TCP/UDP
实时性	低	高
分析能力	高	高
服务器流量计费	高	低
设备端流量计费	高	高

由于物联网的数据特性与互联网不同，因此许多设计要素都要相应地调整。以数据流量为例，现在大多数云计算服务供应商仅对服务器流出数据计费，而对于流入数据流量不计费、不限速。这对于营运公司来说是利好，带宽压力不大。但是其设备端的流量成本压力却非常大！如果用户使用费用高，那么就会影响到销量和使用。

根据 Intel 的车联网数据统计，自动驾驶车辆每天会产生 1.5GB 流量，这个费用就太惊人了！因此，必须针对协议和系统构成做出重大改变，比如改用 UDP 或者数据变化触发发送报文的方式等。尤其是数据变化不那么快的物理量，比如体温或冷链管理，其实完全没有必要采用定时传输方式；可以将温度变化作为异步事件推送给服务器。

9.2.7 系统架构

物联网与互联网迥然不同的数据特性，必然会导致其系统架构上的差异。

服务器架构主要包括操作系统、Web 服务器、数据库服务器和编程框架。在互联网中，LAMP/LNMP 曾经是许多中小网站的启动配置，其中 MySQL 起了很大的作用。但是如果自主开发物联网，这套系统无法平移到物联网应用中做入门级系统架构。除此之外，Web PaaS 平台也无法满足物联网的需求。所以，一些云计算企业开始针对物联网提供专门的 IoT PaaS 服务以补足其性能缺失。实际上，IoT PaaS 的 MQTT 不仅仅解决的是设备接入，它还一举解决了数据流动瓶颈，这才是在物联网架构中引入 MQTT 的重要原因。

数据如流水，笔者经常将服务器设计与水利工程类比：

- 数据从设备采集，并逐层上传到服务器。这就好比涓涓细流不断流入支流，支流入长江，长江流入大海。
- 数据汇集到服务器前端，数据总量比较大，必须使用负载均衡服务器配合多台服务器

来处理数据。水利上也使用宽阔的河道和大型水库来蓄水，减少对下游的冲击，并使用多台闸门来放水。

- 如果有更多数据接入，服务器就需要弹性扩展。这就好比遇到洪峰，必须采用更多闸门泄洪。
- 数据流入后，需要处理、利用、分析、归档等。这就好比水利工程中采用地方水库进行蓄水、发电、取样、灌溉等。
- 如果数据的洪流处理不当，就会发生堵塞，并流失数据等。这就好比水利工程中通常所说的堰塞湖、决口和漫堤等。

虽然自然界的水资源与数据流相比，还有许多不一样的地方，但是在笔者眼里，的确将数据视为流动的资源。最重要的一点是利用数据，解决数据流动的瓶颈，不要被数据所淹没。

所谓系统架构，其核心目的是为了满足一定规模的数据在一定时间范围内处理需求的系统设计。这里面隐含了“规模”和“实时”两个要求。而且这两个要求是一对矛盾。系统的开发从小到大，需要经历一个过程，中间需要针对应用规模进行数次架构优化升级，这样才能够满足物联网的实时响应要求。系统架构要与系统规模相适应：小型系统使用大数据系统架构是一种浪费；而大型系统需要充分利用云计算服务商提供的各种组件来重构系统以满足系统规模的需求。架构师的责任就是在有限预算范围内设计出满足预定范围内数据流动、实时处理和批量归档的系统结构，并保留一定的裕量。

物联网架构需要根据数据的瓶颈进行分析，并利用各种方法解决问题：

- 接入服务器与应用服务器分离；
- 应用服务器与数据服务器分离；
- 使用数据缓存改善数据流转的实时性能；
- 使用服务器集群应对高并发连接；
- 数据库读/写分离；
- 使用反向代理和 CDN 加速网站响应；
- 使用分布式文件系统；
- 使用分布式数据库系统；
- 使用 NoSQL 和搜索引擎；
- 业务拆分；
- 数据库分片；
- 分布式服务和微架构。

在表 9-2 中，列出了 Web 与物联网应用服务器在组件选择上的不同。在系统构成中，主要变化的是代理服务器和数据持久层设计，尤其后者特别重要。因为这有可能是应用的性能瓶颈，也是后续数据端大数据分析的基础。

表 9-2　Web/IoT 组件对比

项　　目	Web	IoT
操作系统	Linux	Linux
代理服务器	Ngnix	Ngnix+MQTT+CoAP
数据持久层	MySQL/SQL	SQL/NoSQL/NewSQL/TSDB
语言框架	PHP/Python/Java/Node.js	PHP/Python/Java/Node.js
数据缓存	Redis	Redis
消息队列	各类 MQ	各类 MQ

此外，网络编程框架的选择也需要留意。在 Web/IoT 数据属性对比中，可以看出其差异点在于：

- 连接协议——HTTP/MQTT/CoAP；
- 连接特性——短连接与长连接；
- 数据流向——流入和流出数据之比；
- 数据实时——流入和流出时间之差；
- 数据分析——数据类型不同。

物联网编程框架要针对特定设备进行协议定制，支持 TCP 长、短连接和 UDP 连接方式；为设备定制符合行业标准的安全接入协议；采用合理的 I/O 复用模型和并发编程模型，以维持设备的高并发连接数，还需要维持与后台数据持久层的连接池。这样才能够实现海量数据从设备到数据库畅通无阻的流动。

9.2.8　数据持久层

数据持久层不会因为互联网/物联网的分野而不同，而是根据不同业务的数据需求而有所不同。理想的数据持久层是，海量数据流入/流出，并持久保持在磁盘中，基于海量数据并在可接收的时间（如 0.1 秒）内进行数据的插入、检索、更新、删除、分页。但是实际上单机数据库根本无法满足“海量”这个标准，更不要说同时满足这些要求了。

设计者需要在数据存储规模和实时性之间根据所选的技术平台进行平衡和妥协。无论是 SQL、NoSQL 还是 NewSQL，都需要系统架构师根据用户需求、数据本身特性进行选择，并进行分层存储和处理。在小型物联网系统中，考虑到一些用户反复提及要实现数据的实时分发和绘图，笔者将实时响应能力作为最高优先级实施。

笔者最初使用 MySQL 单机版数据库进行设备记录和检索。当时，设备数量少于 10 台，每台设备以 0.5 秒间隔发送数据，并在网页中实时绘制数据波形。其需求非常简单，接入数量也非常少。即便如此，数据也在 MySQL 中快速累积，绘图数据从同一数据库表中读取，请求返

回时间逐渐延长。从 0.01 秒到 1 秒，直至 10 秒，速度越来越慢。虽然采用一些数据库优化技巧可以解决部分问题，但一旦设备接入数量上升，问题会重现。这个“坑”让笔者了解到：传统数据持久层方案必须被推翻，并针对物联网的数据流向、更新频率、设备容量重新规划。现在，笔者还在不断地学习、更新这方面的各类方案，但总的原则可以与大家分享：

- 根据数据的特点，采用 NoSQL/NewSQL 及分布式大数据库配合传统 SQL 进行分级存储，代价是增加设计复杂度；
- 更新频率高的数据，可以被视为“热点数据”，其必须通过内存型数据库进行缓存，比如 Redis；
- 热点数据，在单机版中被视为数据快照，用完就丢；
- 热点数据，可以在大数据数据库中进行保存，但需要定时归档；
- 更新频率不高的数据，可以保存在 SQL 结构化数据库中，但需要定时归档，并从数据库中删除不用的数据；
- SQL 数据库用于保存用户信息、统计信息、配置信息，以及保存数据分析结果；
- SQL 数据查询需要分段、设置限制，避免长时间占用数据库连接；
- SQL 数据库定期删除非热点数据，操作时段应选择服务器闲时，以减少锁表给其他用户带来不便；
- SQL 可以根据业务进行切割，必要时可以使用 PaaS 中的读/写分离分布式 SQL 数据库，这比自己实现要靠谱；
- 采用 MQTT 或者消息队列将数据在持久层之前转发给第三方或者数据分析服务，回避了数据库的流量瓶颈；
- 根据数据容量的分析需求，可以选择云计算平台提供的 BigTable、MongoDB、HBase、各类 TSDB 等服务作为大数据库持久层；
- 大数据、BI、人工智能等需要采用对应平台的数据库技术。

笔者提供的 EPIC 系统就是将 Redis、MySQL 和 TSDB 实现了分级存储。对于实时性要求特别高的数据，我们会将其保存在 Redis 中；对于需要持久保存的数据，我们会将其保存在 MySQL 和 TSDB 中。随着对于其他数据库的逐渐了解，以及客户规模的逐渐扩展和需求的改变，我们会考虑采用其他数据库，比如列存储数据库 Cassandra、阿里云的 DRDS 等。但是切换数据库代价巨大，需要我们有更多时间和更多数据去测试。

由于 NoSQL 与 SQL 存在较大差别，因此，NoSQL（如 Redis）在某些场景中固然有着特殊的妙用，但是在某些场景中却也缺乏一些原有 SQL 的特性，比如分页，需要采用其他手段来实施。如何充分利用这些新技术的特性，回避它们的短处，则是考量架构师能力的标杆之一。此外，各类时序数据库是解决物联网时序数据存储的备选方案之一。

9.2.9 大数据分析架构

大数据的特点是体量大、种类多和速度快。一般来说，小型应用会采用数据快照方式保存最近的热点数据。而超出时间窗的数据会进行分析后归档。由于大数据与传统数据库、数据仓库、数据集市存在一些差异，所以需要根据大数据分析的需求来定制数据端的系统架构。一般来说，大数据分析平台架构的层次从高到低如下：

- 各类移动 APP、桌面应用，用于获取数据可视化和管理大数据；
- 大数据应用程序，主要是规划大数据算法等；
- 数据分析，包括传统统计分析和高级的数据挖掘算法；
- 可分析的数据仓库和数据集市；
- 数据库与工具；
- 可操作的数据库，包括结构化、非结构化和半结构化数据库；
- 安全框架，访问控制等；
- 带冗余的物理基础架构。

在目前笔者手头的项目中，没有完全符合“大数据”定义的工程。

9.2.10 业务耦合与分离

许多网站在业务单一的时期，采用的往往都是紧耦合的架构。随着各种业务的不断加入，追求不同组件的高内聚、松耦合架构成为主流诉求。业务逻辑之间的解耦，可以解决业务单独升级维护以及水平规模扩展的需求。

在物联网应用中，笔者也尝试了将单一服务器拆分成设备云和业务云的设计。设备云负责设备的接入和数据转发，业务云负责数字资产管理应用，两者间采用 REST API 进行沟通。这样的做法好处很多：

- 设备云可以独立发展，实现设备的抽象化，与企业业务流程无关；
- 设备云可以有独立的数据缓存与时序数据库，实现从中等规模到超大规模的发展；
- 业务相关的权限管理、资产管理、工作流程之类与企业有关的迭代需求不再影响设备接入，企业可以非常容易地修改权限管理和引入第三方合作伙伴；
- 企业可以非常容易地实现多种类的客户端 APP。

设备云与业务云之间仅有 REST API 还不够，笔者计划实现 JavaScript API，以实现跨域实时绘图、实时分析等方面的功能。

此外，Docker 容器给“微服务架构”点了一把“火”。基于容器的云计算模式被称为容器即服务（CaaS Container as a Service）。原本单体式（monolithic）的设计被分解为可以独立运行的单元，比如负载均衡、云服务器、云数据库、MQTT 接入、CoAP 接入、Redis 数据库、MongoDB

以及消息队列都可以独立存在，而且可以配合 Git 源码管理进行快速配置、多机部署，彼此之间通过消息队列进行异步整合。这就是松耦合的微服务架构。其可以实现使用灵活、运维简单和构建快捷的理想状态。将设备云和业务云分别容器化也是笔者的下一步计划。

9.2.11　业务与数据融合

与第三方合作伙伴分享数据，实现业务和数据融合是一种行业趋势。这种开发源泉来自互联网，物联网必将继承并发展这些实施方法和技术方案。

在传统的架构中，数据上传到 Web 服务器，Web 服务器将数据保存在数据库中，而第三方通过 Web 服务器的 REST API 提取这类数据。但这就造成数据必然流经数据库，容易导致系统性能瓶颈。物联网除了可以重用互联网常见的 REST API/SSE/WebSocket 外，还可以通过 MQTT 实现数据分享。

这里 MQTT 的作用具有多种意义。MQTT 独立于 Web 服务器，数据转发在数据入库之前发生。MQTT 不仅仅解决了数据接入的问题，还减轻了数据库压力，同时附带解决了数据分享和鉴权问题。所以，需要综合地理解 MQTT 在物联网中的流行原因。

物联网还会涉及大数据分析，其持久层采用何种数据库、数据仓库、大数据库保存数据，并在此基础上实现大数据分析和利用，这些都需要预先规划。另外，数据交易采用何种交换形式进行数据交换和变现、如何符合国家法规，需要仔细调研、规划，然后再分步实施。

9.2.12　认证授权与计费

互联网的认证、授权和计费（AAA；即“Authentication，Authorization，Accounting”），无论在开放的互联网服务中还是在封闭的商业应用中，都有不少解决方案。云计算供应商也会提供基于自己生态的解决方案，微软、谷歌、Amazon（亚马逊）、百度、阿里、腾讯等都提供各自的账户 ID 服务。开发者能力强的，也可以在开源的 OpenID、OAUTH、SSO、LDAP、RADIUS 方案基础上自行建设。

物联网设备云、物联网运营商业务支撑等应用，也会充分利用这方面的互联网设计来完善自身安全、访问控制和计费方面的需求。

9.3　物联网网关与边缘服务器

如果将物联网分为设备端和服务器端，则基于 Linux 的物联网网关从网络拓扑上应该归于设备端，但是其在软件架构上却和服务器端类似。

前面介绍了物联网应用系统存在不同的网络拓扑，以及网关和边缘服务器的区别。尤其是

私有网络与公众 IP 网络之间，往往需要物联网网关。在某些雾计算应用中，边缘服务器也是必需的。但在实际工程中，这两者往往运行在同一台物理计算机上：前者负责协议转换，后者负责近场的数据处理。本节将就如何利用 Python 设计此类混合型服务器提供一些建议。

9.3.1 Python socket 服务器

Python 与 Linux 一样，与生俱来就支持互联网。在不使用任何其他库的前提下，其内置的 socket/urlib2 库就可以用来构建简单的 TCP/UDP 和 HTTP 客户端/服务器端程序。

基于 socket 之上已经出现了大量的 Python Web 框架，有 Django 这样的重型框架，也有许多轻量级框架（如 Flask）。这些轻量级框架仅具备基本的路径规划和网页模板，可以用于网关的配置界面、本地数据分析和可视化。网关内置管理页面的同时连接数量较少，所以 Flask 的性能就足够了。但是中心服务器需要考虑并发连接数量，Flask 服务器需要前置代理服务器。

受限于 Python 的 GIL，所以 CPython 在多线程支持多个客户端上有一定困难。这里有若干解决方案：

- 使用其他 Python 实现，如 PyPy、Stackless 等实现；
- 使用没有 GIL 的 Python 实现，如 Jython 虚拟机；
- 使用单进程异步 I/O 的 Python 框架，如 Twisted，而异步 Web 还有 Tornado；
- 使用协程；
- 使用 gevent 突破 GIL 限制，开发者甚至不需要修改原有代码就可以支持异步 I/O。

9.3.2 pyserial RFC2217

设备中最常见的端口是串口或 USB 虚拟串口。pyserial 连接的端口是 RS232/UART，或是 USB 虚拟串口（VCP）。在很多物联网应用中，我们往往需要远程访问这些物理端口。由于串口的数量和传输距离是有限的，因此将串口承载在 TCP/IP 通道之上可以解决这些问题。比如：

- 对于分布在各个地点的离散设备进行远程监控；
- 通过 TCP/IP 通道对机房、ATM、OA 和 BA 系统中的设备进行监控；
- 通过 TCP/IP 扩展多个串口；
- 通过 TCP/IP 来给 MCU 远程下载固件；
- 通过 TCP/IP 将地理位置较远的串口设备进行互相连接。

所谓串口服务器就是此类设备，用于将 RS232/485/422 串口连接上 TCP/IP 网络端口。通过串口，传统设备可以立即具备网络连接能力；通过网络通信，极大地突破了串口通信的通信距离。在 5.5.1.6 节介绍的 pyserial 扩展中包含了一个端口转发的应用例子，即远程登录端口控制（Telnet COM Port Control Option）RFC2217 协议。到目前为止，RFC2217 依然是一种实验性协议。

最近，设计农用无人机的朋友提出了一个需求：通过一台 Android 平板电脑连接多台数传电台以控制多台无人机进行灌溉。笔者一开始考虑了多种硬件解决方案，如 Android USB Host API。由于时间紧迫，笔者给出的建议如下：

- 利用 Linux SBC + pyserial + USB 串口，连接多台数传电台；
- 每个串口映射到不同的端口；
- Android 客户端通过 socket 访问 RFC2217 端口来访问多个电台。

另外一个例子来自设计 Wi-Fi 遥控的朋友，他急需能够立即实施的方案。笔者也给出了同样的技术方案：Linux + pyserial + USB 串口，串口连接远程设备，通过 APP 甚至终端软件对远程设备进行控制。需要升级的也就是设备中的 MCU 固件而已。

所以，在连接数不太多的情况下，pyserial 是一个可以快速实施的方案。

9.3.3　SubGHz 网关 panStamp

panStamp 由西班牙工程师 Luis Miguel Martin 开发，它是围绕 TI/Chipcom 的 CC1101/CC430F5137 的 SubGHz 无线模组产品，面向智能农业市场。图 9-7 是其中一款基于 CC430F5137 的模块产品。其原来仅支持 868MHz，即欧盟 SubGHz ISM 频段，但近来增加了国内常见的 433MHz 频段。通常，此类“小无线”大部分采用私有协议，大多数不开源。panStamp 的作者 Luis 却把所有设计均开源共享。该项目的特点如下：

- 无线模块生产工艺较好，采用沉金工艺，与国内的无线模块相比，更耐腐蚀；
- 采用 F5137 SoC 开发单片方案，固件兼容 Arduino；
- 定义了 SWAP 无线协议，并支持 OTA 固件更新；
- 基于消息队列的松耦合设计；
- 自带 Web 管理界面。

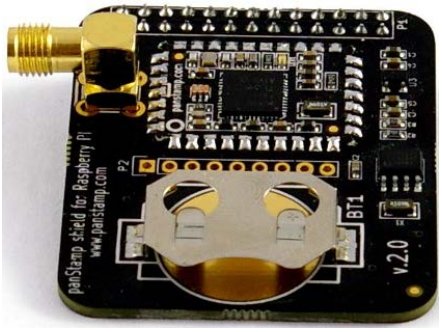


图 9-7　panStamp 模块产品图

Luis 在 panStamp 开发中大量使用 Python，如下所示。

- pyswap：为应用提供 SWAP 无线协议支持；
- swapdmt：SWAP 设备管理工具，GUI 版本；
- swapdmt-cmd：SWAP 设备管理工具，命令行版本；
- lagarto：开源自动化软件，网关主体软件。

Lagarto-SWAP 是一种网关，也是服务器，其与 panStamp 设备进行 SWAP 协议通信，与后端云服务器（如 Xively 等）采用 TCP/IP 网络进行通信，并负责协议转换。其协议转换基于 pyswap 包和其他 Python 标准包。Lagarto-SWAP 提供了类似于无线路由器的配置方式，其网页框架采用了轻量级 Web 框架 CherryPi。lagarto 网络配置项目主要是进程名称、广播通道和服务器端口。

lagarto 中使用了 ZeroMQ 这一消息队列作为不同组件之间的互动桥梁，最近的设计则升级到了 MQTT。这带给笔者的启示很大：即便嵌入式应用（如网关）中也可以通过消息队列实现松耦合架构以简化设计。

在 lagarto 架构中，lagarto-MAX 是关键部件，它是一种事件管理器客户端，用于将属于不同网络的物理量整合起来呈献给用户。此外，它还可以基于时间与网络事件进行可编程的处理。从这一点来看，其类似于 IFTTT 服务器。lagarto-MAX 可以接收、重发、处理来自同一 IP 网络中的 lagarto 服务器事件。

lagarto-MAX 的特性表：

- 接收同一 IP 网络，如 LAN 内网或 VPN 节点的 lagarto 服务器发送的网络事件；
- 向网络中的 lagarto 服务器发送命令；
- 通用的 Web 控制/监控界面；
- 可以通过 Web 界面进行事件管理器编程；
- 自动上传数据到不同的云数据服务；
- 可独立运行的 Python 脚本；
- 管理数据的 Python API；
- 本地数据存储（SQLite）；
- 实时创建 Web 图表。

所以 lagarto 网关不仅仅提供了一个最简单的星型拓扑 WSN，还提供了协议转换、事件处理、管理界面、输出存储、设备联网，以及松耦合的设计架构。所以，这是一个网关与边缘处理混合的设计，具有很高的参考价值。

9.3.4 Rascal micro

Rascal micro 基于 Python、JavaScript 和 HTML，构建 IoT 网关，产品照片参见图 9-8。该项目的 PCBA 售价为 149 美元，与其他开发板相比，这显得比较贵。其官网地址为 http://rascalmicro.

com/index/。

图 9-8　Rascal micro 产品图

Rascal 由一枚 AT91SAM9G20 驱动，内核为 ARM926EJ-S，时钟频率为 400MHz，并提供了 USB 主机、以太网接口和 TF 卡插槽。最重要的是其使用了 Arduino Shield 接口，可以扩展许多传感器和 WSN 接口。简而言之，它是一款由 ARM9 驱动的大号 Arduino。

Rascal 内置 Flask Web Server，它提供了一个网页版的 Python 编辑器，并可以在其内部运行 Python 用户代码。其前端使用 jQuery 和 jQplot。虽然 Rascal 没有指定其设备通信端口，但是 Arduino 有许多有线连接或者无线连接 Shield，通过 USB 也可以支持多种 WSN/BLE 等 Dongle。通过组合 Wi-Fi/BLE/WSN，可以将 Rascal 用作 IoT 网关。

许多 Linux SBC 在设计时都宣传自己兼容 Arduino Shield I/O 定义。比如 UDOO、Arduino Lei、Arduino Yun、Edison、PCDuino 等，这类设计都可以被视为 IoT 网关的备选平台。

9.3.5　Java IoT 网关

Freescale 基于 QorIQ LS1021A，在 Linux 和 Oracle J2SE 基础上设计过物联网网关设计，并采用了 Java 动态模块化框架 OSGi。其系统架构参见图 9-9。可惜除了开发板外，其很少有商品化产品出售。Oracle 官网上提供了如何通过 Java 访问树莓派资源的文章，所以在 Linux 中通过 Java 来实现网关没有任何问题。

拥有 Java VM 对 Python 开发者也是利好。只要是 Linux 主板，即便没有 CPython 环境，通过运行 Jython，Python 网关程序同样可以运行于 Linux+Java 盒子中。所以，基于 Linux SBC，Python 开发者的硬件平台空前广阔：

- LFS/CLFS Linux 最小系统，运行交叉编译的 CPython、MicroPython 或 PyMite 的 UNIX 版本；
- Linux + Java，运行 Jython，并支持安全沙盒；
- Linux + CPython，运行原生 Python 程序。

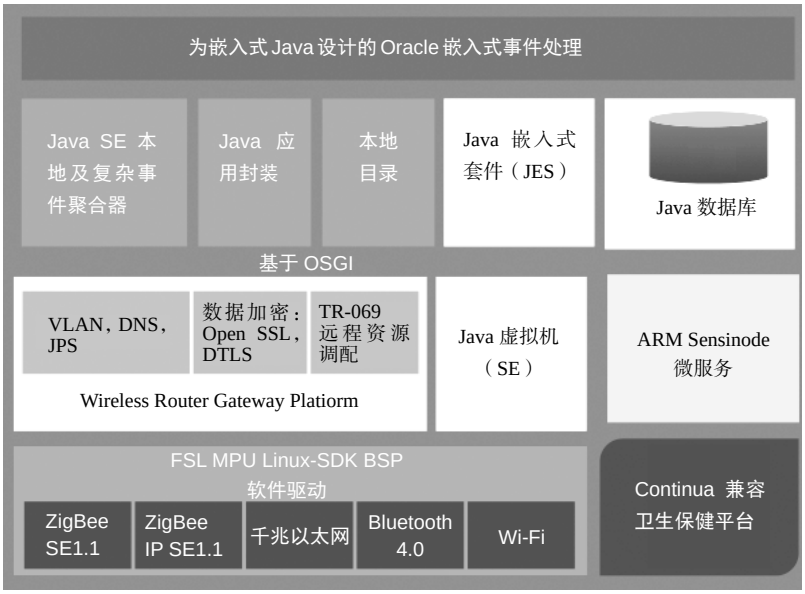


图 9-9　Freescale 的嵌入式 Linux/Java 物联网网关系统架构图

与采用 Java 开发网关相比，采用 Python 和开发类似于 lagarto 的 IoT 网关可降低设计难度。这对于鼓励开发者甚至消费者来对网关进行编程和定制，是一个很好的生态构建途径。

9.4　物联网设备接入协议

远程设备可以通过 4G 网络或者 Wi-Fi 直接连接 TCP/IP 网络，也可通过网关、边缘服务器等设备转换到 TCP/IP 网络，再连接到设备云、应用云、大数据分析服务器和用户 APP。本节内容将总结设备云各种接入协议的实现，包括：

- 基于 TCP/UDP 的套接字服务器；
- 基于 UDP 的 CoAP 协议；
- 基于 TCP 的 MQTT 协议；
- 基于 HTTP 的 REST API Web 服务。

除了这几种，还可以找到其他一些协议：

- HTML5 WebSocket/SSE，可以支持持久连接和服务器推送；
- XMPP，用于即时通信等。

严格地说，应该将后两类定义为互联网客户端接入协议。客户端本质上也是设备端，但是其已经独立发展，形成了自己的特点。

从目前来看，许多设备联网都采用私有协议，需要定制接入服务；物联网专用协议 CoAP/MQTT 也很热门；而基于 Web 的 REST API 和其他方式也是互联网标准。所以，本节将以 Twisted 为例，讲解如何接入这些协议。

9.4.1　异步通信框架 Twisted

Twisted 设计于 2000 年，是事件驱动的异步网络引擎架构，作者是 Glyph Lefkowitz。Twisted 最初是针对一款多人在线游戏（Twisted Reality）而设计的网络通信框架，其后来发展成为一个事件驱动、跨平台的可扩展型通用网络应用开发框架。

Twisted 自推出以来广泛用于生产环境中，Google、Lucas Film、Justin.TV 及 Launchpad 软件协作平台都采用 Twisted 设计过产品和网络服务。此外，Twisted 还驱动着 BuildBot（自动构建工具）、BitTorrent（著名 P2P 分享服务）及 TahoeLAFS（开源云存储服务）。

作为网络应用的重型框架，Twisted 支持常见的传输层和应用层协议，包括 TCP、UDP、SSL/TLS、HTTP、IMAP、SSH、IRC、FTP 和 DNS。用 Twisted 定制开发客户协议也很简单。Twisted 对于其支持的协议都带有客户端和服务器端，同时附有命令行工具和 Log 日志输出，这可以用于配置和部署产品级的网络应用。

图 9-10 来自 Twisted 的参考设计：Poetry Server。Twisted 是单线程设计，运行于单一 CPU 核心中，可以在单一线程中集成多个端口服务。但 Twisted 通过 Reactor 模型构建了高速异步事件驱动型网络编程框架。

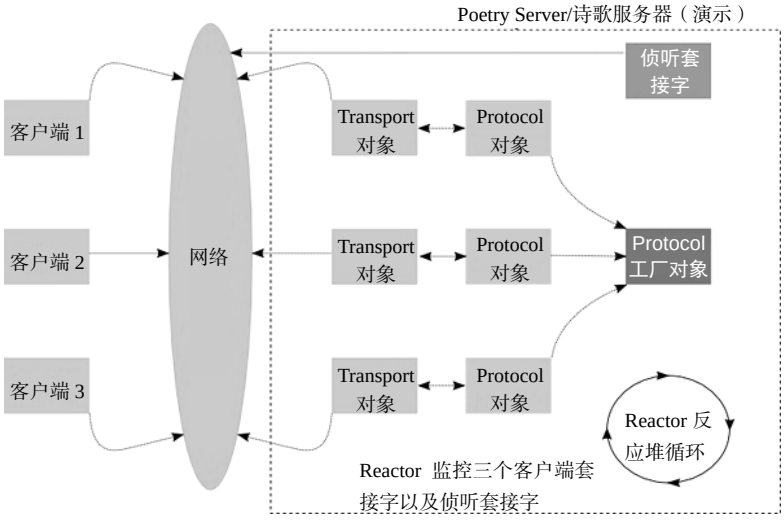


图 9-10　Twisted 系统运行示意图

9.4.1.1 多核服务器与负载均衡

Ngnix 由俄罗斯工程师 Igor Sysoev 开发，其徽标如图 9-11 所示。Ngnix 可以作为 Web 代理服务器、Mail 代理服务器和通用的 TCP/UDP 代理服务器。套接字服务器、Web 服务器、Web API 服务器的负载均衡可以采用 Ngnix 来实现，这几乎是行业标配之一了。

图 9-11 Nginx 徽标

在多核服务器中可以运行多个 Twisted 实例，并可以非常容易地进行规模性能扩展。但一台服务器一般只会分配一个公网 IP。如果 Twisted 实例分别侦听不同端口也没有问题；如果共享一个侦听端口，则需要负载均衡服务器来实现。

作为主流的反向代理服务器，Ngnix 可以支持多台多核高配主机，运行多个 Twisted 实例实现横向规模扩展。具体实现方法如图 9-12 所示。

- Twisted 1，运行于 Server A，CPU1，侦听端口 5000；
- Twisted 2，运行于 Server A，CPU2，侦听端口 5010；
- Twisted x，运行于 Server A，CPUx，侦听端口 50x0；
- ……
- Twisted 1，运行于 Server B，CPU1，侦听端口 5000；
- Twisted 2，运行于 Server B，CPU2，侦听端口 5010；
- Twisted x，运行于 Server B，CPUx，侦听端口 50x0；
- ……
- Ngnix，侦听指定端口 6000，并按照预先定义的分配策略将流量转发给 Server A/B/Z 中的 CPU1/CPU2/CPUx。
- 还可以在 Ngnix 处部署 TLS 证书等。

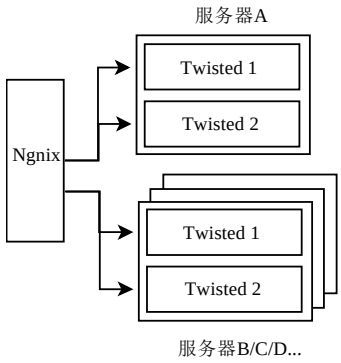


图 9-12 Ngnix 与多个 Twisted 服务器实例构成负载均衡设计

基于 TCP 的四层负载均衡并发能力是 50 万个长连接，这可以满足大多数物联网需求了。如果需要更多连接数，可以通过子域名将流量分配到多台负载均衡的方式实现规模扩展。

9.4.1.2　客户端

在 Twisted 的参考设计中，一般会对等提供服务器和客户端设计。在应用层上，用户还可以自行替换或者扩展 Twisted 的客户端。Web 客户端中较为出名的是网络爬虫框架 Scrapy。

9.4.1.3　服务器端

Twisted 虽然提供 HTTP 端口的客户端和服务器，但是其仅支持简单的 HTTP 开发。基于 Twisted，可以使用更加完整的 Web 框架，如 Klein 或 Cyclone 等。物联网 CoAP 协议对应的代理服务器是 txThings。MQTT 有 twisted-mqtt-client 客户端。笔者曾经误以为 Twisted 方案是实现 MQTT 服务器，但其实它是作为 MQTT 客户端存在的：

- MQTT broker 是服务器，比如开源的 mosquitto，或者云服务供应商的 IoT 套件；
- twisted-mqtt-client 作为 broker 的客户端，挂接在 broker 服务器上；
- twisted-mqtt-client 将数据保存在持久层（如 SQL/NoSQL/NewSQL 数据库服务器）中。

所以在基于 MQTT 的设计中，Twisted 提供的都是客户端设计。但之所以将其归类到服务器，主要因为这都是服务器端领域的设计。

从物联网接口来看，Twisted 已经支持了大多数的设备接入协议：

- REST Web API，基于 Web 服务器；
- Raw TCP/UDP，定制客户协议，用于传统设备升级；
- CoAP，用于超轻量级物联网；
- MQTT，用于轻量级物联网。

与 Twisted 相关联的有 Tornado 和 Cyclone。Twisted/Tornado/Cyclone 的英文含义是类似的。这些开源工程的名称暗示自己的设计运行速度飞快。

这三者间存在联系和差异：

- Twisted 是独立发展的网络编程架构，基于 epoll/poll/select，跨操作系统。
- Twisted 不仅仅针对 Web，它还包括了许多其他网络协议。
- Tornado 是 FriendFeed 设计的异步非堵塞 Web 网络框架，基于 epoll。
- Tornado 虽然也可以支持套接字服务器，但其主要作为 Web 服务器使用。
- Twisted 的 Web 过于简单，Cyclone 项目将 Tornado API 在 Twisted 上重新实现，是 Twisted Web 完整框架。
- Cyclone 的缺陷在于生态和社区不够强，只有作者一人维护。
- Twisted 上还推荐 Klein 做 Web，Klein 比 Cyclone 更加简单，但 Cyclone 更加完整。

虽然 Tornado 也支持 socket/WebSocket/Web Server，社区生态也很强，但笔者希望尽量减少

框架种类，还是采用了基于 Twisted/Cyclone 的完整生态开发。

在服务客户定制项目的过程中，笔者以 Twisted 为基础实现了 EPIC 设备连接服务器，具备以下特点：

- 物联网设备联网使用 Twisted TCP/UDP 套接字服务器；
- Web/Web API 使用 Cyclone；
- MQTT 接入采用 mosquitto 开源服务器；
- CoAP 接入采用 txThings 代理服务器；
- 消息队列采用 Redis/ZeroMQ；
- 负载均衡使用 Ngnix 或 PaaS；
- SQL 数据库采用 MySQL/MariaDB；
- 内存数据库采用 Redis。

9.4.2 Twisted 套接字服务器设计

由于当前许多物联网项目实际上都是将现有产品实现联网升级，而传统设备大多数采用串口通信，因此这些企业很自然地就想将串口协议照搬到TCP长连接上。实际上这是一个“大坑”。

首先，这些协议的底层实现不一样了。串口协议定义之初没有考虑到网络连接的不稳定性，串口接收数据可以通过超时管理来做报文隔断。而 TCP 长连接的特有现象是粘包和半包。解决方法就是缓存、分割、解析报文帧结构。所以，服务器端要专门设计定制协议。这些后面会进一步解释。

其次，此类串口协议往往比较简单，多采用预定义二进制结构传递物理量数值，没有序列化、逻辑层次与逻辑端点概念。一旦增加逻辑端点（如增加若干物理量字段），或增加逻辑功能（如增加下发指令、固件升级、远程授时、密钥升级、算法升级），那么整个协议就必须重新定义。结果，设备与服务器对接很容易落入对接、测试、再升级的怪圈，以致延误很多时间。

再次，当不同协议版本的设备同时联网时，服务器端就会出现调试困难、频繁抛出异常等，各类问题层出不穷。

这些困难对于设备端开发者完全无法理解。而且在项目启动之初，各方都可能无法预见后续的变更。而一旦开始迭代，就是噩梦的开始。所以，系统设计师应该向自己的客户或管理层清楚表达并坚持采用标准化物联网协议进行设备联网。如果不得不采用私有协议，退而求其次，也必须重新审视现有遗留协议是否具备完整的报文结构、逻辑分层、序列化标准。系统设计师必须兼顾协议的设备可实施性和服务器可行性。否则，此类私有协议从工程启动之初就注定成为互相推诿的环节，成为失败的主因之一。

9.4.2.1　基础协议实现

Twisted 是比较重型的网络编程框架，其特点恰恰是 TCP/UDP 私有协议的快速定制，并在此基础上，Twisted 内置支持了许多互联网常见协议。Twisted 的教程中介绍了最简单的 Echo Server 例子：

（1）客户端将信息发送给服务器；

（2）服务器将信息原样返回给客户端；

（3）循环往复。

```
From twisted.internet import reactor, protocol
from twisted.protocols import basic

class EchoProtocol(basic.LineReceiver):

    def connectionMade(self):
        print “Connection from ”, self.transport.getPeer().host

    def lineReceived(self, line):
        if line == ‘quit’:
            self.sendLine(“Goodbye.”)
            self.transport.loseConnection()
        else:
            self.sendLine(“You said: ” + line)msg

class EchoServerFactory(protocol.ServerFactory):
    protocol  = EchoProtocol

if __name__ == “__main__”:
    port = 5001
    reactor.listenTCP(port, EchoServerFactory())
    print “Server running on %d, press ctrl-C to stop.”%(port)
    reactor.run()
```

这个例子虽然简单，但却是应用层通信协议的基础。完整协议还应该包括传输层安全和基于应用层的序列化标准。如何定义一个兼顾安全性、设备与服务器端实施难度以及容易运维、升级与调试的通信协议，需要系统工程师预先了解系统需求，仔细规划，谨慎实施，切勿轻视。

9.4.2.2　TCP 与 UDP 协议

在 Linux 中，将 TCP 通信称为“Stream”数据流，将 UDP 通信称为“Datagram”数据报。物联网联网协议定制，必须首先确定是 TCP 长连接、TCP 短连接还是 UDP 协议。这三种方式各有优缺点。9.4.2.1 节中介绍的是基于 TCP 长连接的文本协议。

TCP 数据流面向连接，尤其是长连接，带来的问题是报文之间的切割问题。TCP 短连接可

以解决这个问题，但是它会带来更多与连接相关的系统消耗。

物联网常见的实施方法是 TCP 长连接，因为长连接可以穿透内网，持续不断地将所采集的数据上传给服务器，服务器也可以将下行数据推送给设备。但是接收端会出现连续接收到报文（粘包）以及只收到部分报文（半包）的情况，接收端实现缓冲区、拼接并切割出正确的报文，而这些算法的实施难度与报文封装格式有关。下面会着重解释此类现象的原因与解决方案。

UDP 数据报不面向连接，比 TCP 连接实现更简单，但需解决数据丢失、重复收发以及推送下行数据问题。由于不同数据的路径有所不同，发生的丢包、重发、顺序颠倒问题，都需要服务器端进行排序、流量控制处理。UDP 还有一个问题是无法穿透内网。

9.4.2.3 序列化标准

在确定了传输层技术之后，需要选择序列化标准。序列化协议可以分为文本与二进制两大类。根据笔者的实际工程实践，文本协议的好处是开发周期快，升级维护容易。最典型的文本类型序列化标准就是 JSON，在服务器端实现起来很容易，可以很容易地映射到 Python dict 类型。举个例子：

```
{“name”:“smartie”, “rom”:“64KB”, “ram”:“8K”}
{“name”:“smartie”, “rom”:“64KB”, “ram”:“8K”, “core”:“cortex-m0”}
```

这两个 JSON 字符串传输后，即使接收方不认识 core 属性，也可以直接忽略，不会造成系统异常或崩溃。此外，文本协议大部分采用回车或者 0x00 作为结束符，报文切割起来很容易。文本协议易用性的代价是带宽利用率低。

二进制协议的带宽利用率高，可以接近 100%。和 JSON 之类的文本协议不同，如果增加或者删减一个字段，通信双方都需要升级协议，这在迭代速度和维护成本上处于劣势。如果读者对于带宽不是很计较的话，推荐采用文本协议。

1．粘包和半包

TCP/IP 传输的特点是不稳定，端到端的时间是不确定的。假设一个数据帧的结构是“HLPE”（即 Header|Length|Payload|End），由于网络传输的原因，到达接收节点时，发生接收到前后连续两帧的现象：“HLPEHLPE”。这就是粘包。与此相对应的是，如果只收到部分数据，如“HLP”，则为半包。这种现象在局域网中做测试时不明显；而在公网服务器中，收发两端之间经过的节点越多，此类现象出现得越频繁。在 TCP 长连接情况下，粘包和半包现象是常态。

为了解决此类问题，必须定义一个完整的帧结构，通常包括报文头、长度、负荷、校验码和结束符等。由于头和结束符往往定义了特定的二进制组合（例如 0x02/0x03），因此负荷数据中如果遇到与头和结束符一样的字符 0x02/0x03，还需要添加转义 0x1B，构成 0x1B/0x02 或 0x1B/0x03。转义符本身也需要转义构成 0x1B/0x1B。所以，一个设计完善的二进制协议还挺复杂的。这也是笔者现在更加偏向于采用文本协议，即便采用二进制协议也推荐采用 Google 的

Protocol Buffers 协议来实现通信的原因。这也符合 Python“不重复造轮子”的哲学理念。

Twisted 最初是为了文本在线游戏而设计的，所以它提供的协议以基于文本的居多，比如 IntString 和基于 CRLF 的协议。但是这些协议都不一定适用客户项目，所以我们还是需要推荐一些二进制协议的参考实现。

在 Twisted 文档中，建议将参数保存在定制协议的工厂（Factory）实例中，而非协议（Protocol）实例中。但是仔细研读文档的含义是，需要跨连接的参数要保存在工厂实例中，因为连接断开后，与连接相关的数据将遗失。这句话的另一种理解是，和当前 TCP 连接有关的参数可以保存在当前连接实例中。所以，TCP 报文可以缓存在连接实例的缓冲区中，继而与下一报文进行拼合。另外一种方式就是单独编写一个客户化的协议类，将字节流缓存拼合算法在协议类中实现。

实际工程中的二进制数据通信协议具备一定的帧结构，并由设备和服务器两端的有限状态机控制连接状态。这些协议往往来自设备供应商之前的串口点对点协议。当这些设备升级到物联网时，我们可以参考以下资源来设计帧结构：

- SDLC/HDLC 帧结构，来自 IBM；
- STX/ETX 帧结构，来自 AVRLib 开源工程。

参考 HDLC/STX/ETX 协议，可以在串口和 TCP/IP 之上采用与表 9-3 所列推荐帧结构类似的报文结构。

表 9-3　二进制通信协议推荐帧结构

英　语	中　文	推荐值	是否必需	说　明
Header	头部	0xFF02	必选	因为 TCP 是基于数据流（Stream）的，需要边界判断依据
Length	长度	0x0000	必选	因为这是双方预留缓冲区大小的依据
Status	状态	0x00	可选	传输控制，如 OK/NOK 重传等。这在 UDP 等协议中有意义，而对于 TCP 和串口点对点无意义
Type	类型	0x00	可选	用于区分不同种类的数据包
Payload	负荷	0x00	必选	这是用户数据
CRC	校验和	0xFFFF	可选	用于传输校验和信道可靠性检测
Term	终止符	0xFF03	可选	用于判断传输报文结束
Escape	转义符	0x1B	可选	针对 Header/Term 的转义（Escape）

附加说明如下。

- Header：0xFF02。没有采用 0x0002，是因为考虑到 0x00 在某些语言中代表 Null。
- Term：0xFF03。没有采用 0x0003，是因为考虑到 0x00 在某些语言中代表 Null。
- Length：计算范围从 Status 到 Payload。
- CRC：XOR，初值 0xFFFF，计算范围从 Status 到 Payload。

- Escape：0x1B。

这个推荐帧结构结合了一些常见结构设计。笔者暂时将其命名为TLV格式。在许多协议中，Header/Length/Term 采用 Byte，但笔者觉得 Word 更好，这可以减少转义和支持长度超过 255B 的情况。在 Payload 中，可以使用标准序列化协议，如 JSON/BSON 等。

一些开发者推荐在安全的长连接上删除头/尾分隔符，而直接采用 LV，即采用长度加负荷数据的方式，以避免转义符所导致的问题。在这方面读者可以自行与 TLV 方式一起做些压力测试（压测）对比，做出自己的选择。

针对以上推荐的通信报文结构，笔者提供了一个 message 解码包，作为其他序列化包的底层基础。

message.py：

```
#encoding: utf-8
import struct
import binascii
from datetime import datetime
import time

class _NotEnoughtFrame(Exception):
    pass

class _InvalidFrame(Exception):
    pass

'''
Basic TLV (Type:Length:Value) protocol with more attributes

Header: 2B, 0xFF02
Status: 1B, 0x00
Type:   1B, 0x00
Length: 2B, 0xFFFF
Payload:1B, 0xFF
CRC:    2B, 0xFFFF
Term:   2B, 0xFF03
ESC:    1B, 0x1B

'''
class Message(object):
    format_headers = struct.Struct("!2s1s1s2s") # Header + Status + Type + Length
    format_tails = struct.Struct("!2s2s") # CRC + Term
    sz_headers = format_headers.size
    sz_tails = format_tails.size
    min_sz = sz_headers + sz_tails
    max_sz = min_sz + 2 ** 16
```

```
status_tokens = (b‘\x80’,b‘\x81’,b‘\x82’)
type_tokens = (b‘\xF0’,b‘\xF1’,b‘\xF2’)

hdr_token = b‘\xFF\x02’
end_token = b‘\xFF\x03’


def __init__(self):
    pass

def getType(message):
    if message[3] in type_tokens:
        return message[3]
    else:
        pass # or raise exception

def getStatus(message):
    if message[2] in status_tokens:
        return message[2]
    else:
        pass # or raise exception

def repyToBytes(self, error, data):
    length = chr(len(data)/255)+chr(len(data)%255)
    return self.format_reply.pack(
            b‘\xFF\x02’,error,b‘\x00’,length,data,b‘\xFF\x03’
        )

@classmethod
def fromBytes(cls, octets):
    [header, status, _type, length, payload, crc, term] = cls.format.unpack(octets)
    assert header == b"\xFF\x02"
    assert term == b"\xFF\x03"
    return cls(realtimeData, extra1, extra2)

@classmethod
def parseStream(cls, streamBytes):
    messages = []

    while len(streamBytes) >= cls.min_sz:
        _posHdr = streamBytes.find(cls.hdr_token)
        _posEnd =  streamBytes.find(cls.end_token)
        Print "debug_message, hdr:%d, end:%d, size:%d"%(_posHdr, \
            _posEnd, len(streamBytes))

        if _posHdr == -1:
            # No complete packat at all
            return messages, b''
```

```
            elif _posEnd == -1:
                return messages, streamBytes
            else:
                # Check valid status and type tokens
                if streamBytes[_posHdr+2] in cls.status_tokens and \
                        streamBytes[_posHdr+3] in cls.type_tokens:
                    # Try to parse datalen attribute in packet
                    data_len = struct.unpack('!H',\
                                    streamBytes[_posHdr+4:_posHdr+6])[0]
                    print "debug_message, data_len",data_len
                    data = streamBytes[_posHdr+6:_posEnd-2]
                    print "debug_message, data_size",len(data)
                    messages.append(data)
                    streamBytes = streamBytes[_posEnd+2:]
                    continue
                else:
                    return messages, b''

        return messages, streamBytes
```

使用 Message 类很容易，将接收到的报文字节串/字符串传递给 Message 的实例，返回值是实例根据帧结构解析后的负荷数据列表和剩余的字节串、字符串。新收到的数据流再添加到剩余的字节串和字符串即可作为新一轮解析的基础。以下是各种测试数据流。

message_test.py：

```
from message import Message
import binascii
import random


def getRand(sz=255):
    r = []

    for i in range(sz):
        r.append(chr(random.randint(0,255)))

    #print "rand:",repr(r)
    return ''.join(r)

def getRandStream(x=255):
    r = getRand(x)
    return '\xff\x02\x80\xF0\x00'+chr(len(r))+r+'\x00\x00\xff\x03'

stream1 = '\xff\x02\x00\x01\x00\x00\x00\x00\xff\x03'
stream2= '\xff\x02\x80\xF0\x00\x00\x00\x00\xff\x03'
stream3= '\xff\x02\x80\xF0\x00\x01\xAA\x00\x00\xff\x03'
stream4= '\xff\x02\x80\xF0\x00\x02\xAA\xCD\x00\x00\xff\x03'
```

```
stream5 = getRandStream(16)
stream6 = getRandStream(32) + getRandStream(64) + getRandStream(128)
stream7 = getRandStream(random.randint(0,255)) + \
    getRandStream(random.randint(0,255)) + \
    getRandStream(random.randint(0,255))
stream8 = getRand() # Full random

streams = stream1,stream2,stream3,stream4,stream5,stream6,stream7,stream8

msg = Message()

for buf in streams:
    print "debug_test:\n",binascii.hexlify(buf).upper()
    msglst,remain =  msg.parseStream(buf)
    for x in msglst:
        print "debug_test:\n",binascii.hexlify(x).upper()
    print "debug_test, remain:\n",remain
```

以上代码还缺乏 CRC 校验，因为 CRC 计算式也是需要定制的。此外，测试数据流类型也不够完整。其虽然增加了全随机数测试，但是还需要有针对性地对数据流中包含数据报头、报尾的特殊数据进行测试。此外，这个测试方法是一个比较原始的测试方式。这些都留给读者自行补充吧。在 Twisted 中使用该类很简单，在 dataReceived 函数中调用即可。

```
class CustomTCP(protocol.Protocol, TimeoutMixin):

    def __init__(self, factory):
        self.msg=  Message()
        self.buf = ''

    def dataReceived(self, data):
        # balabala code
        messages, self.buf = Message.parseStream(self.buf + data)
        for msg in messages:
            # parsing, saving
            self.process(msg)
```

帧结构的实施目的就是作为报文消息判断和截取的依据，有了完整的帧结构，就可以从 TCP 数据流中将报文切割出来，并将切割剩余的数据流作为下一次的拼接头部。这样，我们可以实现对应的套接字服务器。

在实际工程中，报文切割后，还需要解析报文，反序列化，对数据进行多维数据统计和分析，以及报警和保存。由于 Twisted 是异步设计框架，因此每个环节都需要是异步设计，尤其是数据保存必须采用异步设计。Twisted 的完整应用太长，以上代码仅作为解释目的。

2．仿真与压力测试

对于开发中的套接字服务器，需要单独的模拟设备与实际的设备连接服务进行交叉测试。

这样可以很容易定位错误，避免出现团队内扯皮的现象。在这种场景下，使用 Python socket 设计虚拟设备来上传数据到服务器。在单一线程中使用 socket 包，开发和调试非常容易。而且上面编写的 message 模块可以复用在模拟设备中。

如果需要压力测试，可以使用 Twisted 客户端来模拟多台设备实现空载以及全业务压力测试。压力测试可以在多台高配计算机上进行，也可以临时租赁几台云主机实施。

前面提到的局域网中的粘包和半包问题，在这里不太容易出现，可以通过由虚拟设备在报文传输中插入延时来实现模拟网络延时情况。

9.4.2.4 Protocol Buffers

二进制协议除了以上的类似 HDLC 的协议，还有 Google 的 Protocol Buffers（简称 protobuf，也有写成 protocolbuf 的）。Google 内部大量使用了 Protocol Buffers 来定义不同通信协议，用于结构化数据和字节码之间的序列化与反序列化。所以，如果能够在套接字服务器中采用该协议，则可以大大简化客户协议的定制。

我们可以粗略地把 protobuf 理解为基于二进制的对象序列化和反序列化的手段，应用层开发者无须钻研底层二进制传输中的隔断、转义、缓冲等细节，只需要考虑哪些对象需要写入和读取即可。实际上它采用每个字节的 MSB 作为缓冲区标识位，和应用层字节没有一一对应。这让笔者联想起 Arduino 里的 Firmata、GSM 短信中的 7 位协议来。这三者有着类似的设计逻辑。但是 Google 为不同语言（包括 C/C++、Java 和 Python）实现了许多现成的包，实施起来可以减少调试和适配的工作量。

1．软件安装

首先需要从 GitHub/Google Code 下载它并安装到系统路径中。其步骤如下：

（1）安装编译器；

（2）在 proto 文件中定义消息格式；

（3）用 protoc 编译 proto 消息文件；

（4）在 python 中使用 protobuf API 读/写消息。

1）安装 protobuf 编译器

源码安装：

```
./configure --prefix=/usr/local/protobuf
make
make check
make install
```

配置路径：

```
vim /etc/profile
添加：export PATH=$PATH:/usr/local/protobuf/bin/
```

```
添加：export PKG_CONFIG_PATH=/usr/local/protobuf/lib/pkgconfig/
```

配置动态库：

```
vim /etc/ld.so.conf
添加：/usr/local/protobuf/lib
```

2）安装 Python protobuf 扩展包

```
cd python
python setup.py install
```

3）Ubuntu 简单安装

更加简单的是采用 apt-get 和 pip 进行安装。

```
sudo apt-get install protobuf-compiler
pip install protobuf
```

2．对象建模

首先我们需要定义 protobuf 的数据结构，保存为 device.proto 文件。

```
package demo;

message device {
    required int32 id=1;
    required string snr=2;
    required string appid=3;
    required string key=4;
    repeated bytes value=5;
}
```

这里需要解释一下 protobuf 的基本数据结构和限定修饰符。其基本数据结构的具体含义可参阅表 9-4。

表 9-4　protobuf 基本数据结构

数据类型	描　　述	封装长度
bool	布尔类型	1B
double	64 位浮点数	*N*
float	32 位浮点数	*N*
int32	32 位整数	*N*
uint32	无符号 32 位整数	*N*
int64	64 位整数	*N*
uint64	无符号 64 位整数	*N*
sint32	32 位整数，高效处理负数	*N*

续表

数据类型	描　　述	封装长度
sint64	64 位整数，高效处理负数	*N*
fixed32	固定 32 位整数	4
fixed64	固定 64 位整数	8
sfixed32	固定 32 位整数，高效处理负数	4
sfixed64	固定 64 位整数，高效处理负数	8
string	ASCII 字符串	*N*
bytes	多字节字符串，如 unicode	*N*
enum	用户自定义枚举	*N*
message	用户自定义消息	*N*

注意

（1）*N* 代表可变，*N* 的长度与具体数值有关联，比如 0~127 的整数的长度为 1B。

（2）enum 打包方式类似于 int32。

（3）fixed32/int32 的区别在于分别针对处理效率和存储效率进行优化。

限定字符串

- required，必需字段，收发双方都必须支持的字段；否则容易出现解码异常，导致报文丢失。
- optional，可选字段，常用于协议升级。接收方如果无法识别，则忽略该字段，正常处理消息的其他字段。
- repeated，与 optional 类似，但一次可以包含多个数值，可以理解为对象类型的数组。

3．代码生成

利用 Google 提供的 protobuf 编译器编译出 Python 类库，然后在 Twisted 中导入并继承 protobuf 的类。

```
protoc --python_out=./ device.proto
```

protoc 编译会在当前路径中产生一个 device_pb2.py 文件，然后用户脚本就可以使用自定义的 protobuf 协议进行通信了。

4．类库引用

我们以下面的演示程序实现 protobuf 的序列化和反序列化。对于网络编程来说，原理是类似的。

protobuf_demo.py：

```
#!/usr/bin/env python
```

```
#encoding: utf-8

import device_pb2
import binascii

dev = device_pb2.device()
dev.id=123456
dev.snr="ENNO2016080"
dev.appid="QazWsxEdc"
dev.key="IjnUhbYgv"
dev.value.append(b'A')
dev.value.append(b'B')
dev.value.append(b'C')
dev.value.append(b'D')

dev_s = dev.SerializeToString()
print "**** Serialization of Protobuf ****"
print "dev:\n%s"%(repr(dev))
print "hex:\n%s"%(binascii.hexlify(dev_s).upper())
print "raw:\n%s"%(dev_s)


decode = device_pb2.device()
decode.ParseFromString(dev_s)
print "**** Deserialization of Protobuf ****"
print "obj:%s"%(repr(decode))
print "**** attributes ****"
print decode.id
print decode.snr
print decode.appid
print decode.key
print decode.value
```

我们在终端可以看到 protobuf 序列化后的字节串内容如下：

```
hex:
08C0C407120B454E4E4F323031363038301A0951617A5773784564632209496A6E5568625967762A014
12A01422A01432A0144
```

9.4.2.5　MessagePack

MessagePack 简称 msgpack。这是一种类似于 JSON 的序列化和反序列化方法。其比 JSON 要小巧，并采用二进制封装。msgpack 的作者是日本的 Sadayuki Furuhashi，他还开发了不少其他开源工程。

使用 msgpack 可以跨平台在多种语言之间交换信息。较小的整数可以在单字节中传输，而短字符串仅需要额外的一个字节即可传输。现在，msgpack 已被五十多种语言和应用软件所使用。

```
{"compact":true, "schema":0}
\x82,\xA7,compact,\xC3,\xA6,schema,\x00
```

以上是 JSON 转化为 msgpack 之后的表达结果。

msgpack 的基本概念是“类型系统”和“格式”。其中 msgpack 的“类型”包括了常见的数据类型，其类型如表 9-5 所示。

表 9-5　msgpack 类型

名　　称	类　　型	限　　制
Integer	整数	范围：$-(2^{63}) \sim (2^{64})-1$
Nil	为空	
Boolean	布尔量	
Float	IEEE 754 双精度浮点数，包括 NaN 和 Infinity	
Raw String	扩展 Raw 类型的 UTF-8 字符串	最大字节数为$(2^{32})-1$
Raw Binary	扩展 Raw 类型的字节数组	最大字节数为$(2^{32})-1$
Array	代表对象序列	最大元素数量为$(2^{32})-1$
Map	代表对象键值对	最大数量为$(2^{32})-1$
Extension	应用自定义的类型	类型信息与字节数组的元组

所谓格式，即采用二进制码来定义不同的类型。具体的规格书可以参考本章延伸阅读部分 GitHub 中 msgpack 的说明。Python 中有 msgpack 扩展包 msgpack-python。

1．msgpack-python

可以使用 PyPI 官网安装 msgpack-python，但是在 Windows 下需要安装 Visual C++编译器。这也说明，msgpack-python 使用 C 扩展进行加速，其执行速度是 JSON 库的 10 倍。

msgpack_demo.py：

```
#!/usr/bin/env python
#encoding: utf-8

import msgpack

x = range(10)
s = msgpack.packb(x)
print repr(s)

d = msgpack.unpackb(s)
print repr(d)
```

运行结果如下：

```
'\x9a\x00\x01\x02\x03\x04\x05\x06\x07\x08\t'
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2．Twisted mspack

msgpack-python 独立于通信框架。但 GitHub/PyPI 上有定制完毕的 Twisted msgpack RPC 扩展包 txmsgpack/txmsgpackrpc。

在 PyPI 官网中有针对 msgpack 和 Twisted 的完整 RPC 测试代码，可以在本章延伸阅读部分中找到对应的网址。

9.4.2.6　其他传输格式

除了 protobuf 和 msgpack，还可以找到大量开源的序列化协议：

- BSON，一种二进制 JSON；
- BJSON，另外一种二进制 JSON，被 MongoDB 采用；
- Thrift，Facebook/Apache 定义的 RPC 通信。

注意　BSON/BJSON 是两个不同的规格。

9.4.2.7　多端口单实例服务器

一般服务器仅仅侦听单一端口，比如 Web 侦听 80 端口，SSH 侦听 22 端口。但在实际工程中，需要在单一服务器实例中侦听多个端口，以实现多种应用目的。在笔者设计的 EPIC 服务器中，实现了以下端口：

- 业务侦听端口，如 5000。
- 消息队列端口，实现发布/订阅（Pub/Sub）模式，其中订阅是需要侦听的服务。
- 业务配置端口，实现运行时配置，而非启动时配置。

除业务端口之外，还增设了一个配置端口，可以用于配置服务器参数，如调试模式、日志过滤及其他等。当然，这个端口需要“严格”保护。如果采用 TLS+密码保护该端口，那么该端口可以面向公网公开。若没有 TLS 保护，则可以不对公网公开，而只针对本机（localhost/127.0.0.1）开放。管理员在通过 SSH 端口登入系统后，可以通过本机配置端口对服务器进行配置管理。

在 Twisted 中可以很简单地配置两个端口。以下代码中就建立了两个端口：

```
#!/usr/bin/env python

import os,sys
import binascii
from twisted.python import log
from twisted.internet import reactor, protocol
from message import Message

class CmdProtocol(protocol.Protocol, TimeoutMixin):
    def connectionMade(self):
```

```
            pass
        def dataReceived(self, data):
            pass

    class MyProtocol(protocol.Protocol, TimeoutMixin):
        def connectionMade(self):
            pass

        def dataReceived(self, data):
            pass

    class MyProtocolFactory(protocol.Factory):
        def buildProtocol(self, addr):
            return MyProtocol()

    class CmdProtocolFactory(protocol.Factory):
        def buildProtocol(self, addr):
            return CmdProtocol()

    print "Test socket server is running...."
    log.startLogging(sys.stdout)
    reactor.listenTCP(8000, MyProtocolFactory())
    # only valid for localhost.
    reactor.listenTCP(5000, CmdProtocolFactory(), interface="127.0.0.1")
    reactor.run()
```

9.4.3 物联网专用协议

CoAP/MQTT 都是物联网专用协议。这两种协议与传统的二进制私有协议、HTTP 等一起完善了物联网的云端 IP 接入协议。

协议选择的考虑点主要是与旧接口的兼容性、标准兼容性以及设备协议栈的支持能力和处理能力。从表 9-6 中可以发现：从设备端来看，两者并没有特别的差异。但在服务器实施上却存在差异。两者虽然有部分重叠的应用，但是存在较大差异。

表 9-6 MQTT 和 CoAP

对比项	MQTT	CoAP
开放度	标准协议	标准协议
目标	资源受限设备	资源受限设备
通信方式	异步通信	异步通信
底层基础	TCP 长连接	UDP
报文长度	短	非常短

续表

对比项	MQTT	CoAP
普及度	更多实现	较多实现
特性	提供灵活的通信模式，二进制数据管道	提供 Web 交互性
模型	发布/订阅模型实现 1：1、1：*N*、*N*：1 关系	1：1
术语	消息（message）/主题（topic）	报文（packet）
通配符匹配	有	无
QoS	最少一次/最多一次/恰好一次	可确认/不可确认
持久层	有，但仅保存最新消息	基于后端 HTTP/REST
安全	SSL/TLS	D-TLS，支持 RSA/AES 或 ECC/AES
其他物理层	TCP/IP	TCP/IP 或短消息 SMS 等基于 packet 的方式
限制	TCP 长连接以及长字符串	NAT 内网穿透需要服务器和路由器配置
数据格式	预定义	资源观察、资源发现
场景	1：*N* 的实时数据总线	适合做状态转换模型
数据流向	设备客户端到代理服务器（broker）	客户端/服务器端双向收发数据，需要采用 NAT 隧道和端口转发
开源工程	paho/mosquitto	Contiki

CoAP 基于 UDP，适合资源非常受限的设备。虽然是 UDP 协议，其设计思路却来自 REST/HTTP。由于 UDP 不是面向连接的协议，因此 CoAP 增加了一些额外的连接相关指令。从服务器设计角度来看，CoAP 可以映射到传统 HTTP 路径中。可以采用 CoAP 代理服务器或者在 Web 服务器中增加 CoAP 协议的方式实现服务器，软件架构改变不大。也就是说，传统 Web 站点增加 CoAP 代理服务器，或者在 Twisted 这种底层框架中增加 CoAP 协议端口都可以。

OMA（Open Mobile Alliance）组织定义的 LWM2M 协议就是承载于 CoAP 协议之上的物联网协议。除了 UDP 数据报服务，CoAP 还可以使用 SMS 短消息作为传输层。

MQTT 基于 TCP，也适合资源受限设备。但是和传统 Web 架构有些不同，其更加类似于即时消息的架构。MQTT 采用的是一种发布/订阅的模式。这种模式原来常用于消息队列这一中间件，主要用于解决大型网站内部不同服务之间异步通信的需求，其也被称为消息总线。使用场景包括耗时批量处理、电邮发送、服务器间的异步进程通信等。IBM 将该组件用于物联网的设备接入目的，其被称为 MQTT。MQTT 前置于其他服务器，类似于代理服务器。实际上，许多消息队列（如 ActiveMQ、RabbitMQ、HiveMQ 等）就直接支持 MQTT。

MQTT 本质上更像高速数据转换开关，但 MQTT 的数据持久层和传统的数据库有些不同。mosquitto 是开源 MQTT broker，采用 C 编写，其数据持久层采用了内存队列加上本地的 DB 文件保存。所以，如果需要将远端设备上传的数据保存下来，需要一个订阅该设备的客户端将数

据保存在数据库中。以 Twisted 为例，需要运行一个 MQTT 客户端订阅设备数据，并连接数据库保存。Twisted 增加 CoAP 端口与增加 MQTT 客户端配合 Twisted 的实现逻辑有所不同。

采用 MQTT 可以非常容易地实现一对多、多对多的数据分享，而无须流经数据库。相比之下，CoAP 也可以通过组播（multicast）实现一对多的数据传输，但一般组播是在同一网段内实现的，或者通过 VPN 跨越公网实现传输。

9.4.4 CoAP

CoAP 为 Constrained Application Protocol 的缩写，它是 IETF 的 CoRE 工作组提议的物联网协议。其目的是为资源“极度受限”的设备提供联网协议。设备可以作为服务器使用，这一点与传统的设备作为客户端应用方式有区别。CoAP 也是代理服务器（proxy），但与 MQTT 不同的是，CoAP 没有内置消息队列，而是通过与 HTTP REST 路径对应的方式提供代理服务。所以，我们可以简单地将 CoAP 理解为基于 UDP 的 REST 服务。

9.4.4.1 Copper 浏览器插件

由于与 REST API 路径对应，因此利用浏览器对 CoAP 进行开发是一件很顺理成章的事情。在 Firefox 扩展插件市场中可以找到一个开发工具插件 Copper。开发者如果有过网页前端开发经验，就会以类似的开发技巧进行 CoAP 代理服务器的开发和调试。

9.4.4.2 CoAP 与 txThings

作为 Python 中的重型网络编程框架，Twisted 的 CoAP 代理服务器已经开发出来了，即 txThings，其作者 Maciej Wasilak 是一位波兰工程师。txThings 工程演示样本包括了 Twisted 作为服务器和客户端的多种设计：

- client_get.py；
- client_put.py；
- client_observer_block.py；
- resource_discovery.py；
- server.py。

我们先看看服务器的设计吧，参见 server.py：

```
import sys
import datetime

from twisted.internet import defer
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor
from twisted.python import log
```

```
import txthings.resource as resource
import txthings.coap as coap


class CounterResource (resource.CoAPResource):
    """
    Example Resource which supports only GET method. Response is a
    simple counter value.
    Name render_<METHOD> is required by convention. Such method should
    return a Deferred. If the result is available immediately it's best
    to use Twisted method defer.succeed(msg).
    """
   #isLeaf = True

    def __init__(self, start=0):
        resource.CoAPResource.__init__(self)
        self.counter = start
        self.visible = True
        self.addParam(resource.LinkParam("title", "Counter resource"))

    def render_GET(self, request):
        response = coap.Message(code=coap.CONTENT, payload='%d' % (self.counter,))
        self.counter += 1
        return defer.succeed(response)


class BlockResource (resource.CoAPResource):
    """
    Example Resource which supports GET, and PUT methods. It sends large
    responses, which trigger blockwise transfer (>64 bytes for normal
    settings).
    As before name render_<METHOD> is required by convention.
    """
    #isLeaf = True

    def __init__(self):
        resource.CoAPResource.__init__(self)
        self.visible = True

    def render_GET(self, request):
        payload=" Now I lay me down to sleep"
        response = coap.Message(code=coap.CONTENT, payload=payload)
        return defer.succeed(response)

    def render_PUT(self, request):
        print 'PUT payload: ' + request.payload
        payload = "Mr. and Mrs. Dursley"
        response = coap.Message(code=coap.CHANGED, payload=payload)
```

```
        return defer.succeed(response)


class SeparateLargeResource(resource.CoAPResource):
    """
    Example Resource which supports GET method. It uses callLater
    to force the protocol to send empty ACK first and separate response
    later. Sending empty ACK happens automatically after coap.EMPTY_ACK_DELAY.
    No special instructions are necessary.
    Notice: txThings sends empty ACK automatically if response takes too long.
    Method render_GET returns a deferred. This allows the protocol to
    do other things, while the answer is prepared.
    Method responseReady uses d.callback(response) to "fire" the deferred,
    and send the response.
    """
    #isLeaf = wTrue

    def __init__(self):
        resource.CoAPResource.__init__(self)
        self.visible = True
        self.addParam(resource.LinkParam("title", "Large resource."))

    def render_GET(self, request):
        d = defer.Deferred()
        reactor.callLater(3, self.responseReady, d, request)
        return d

    def responseReady(self, d, request):
        log.msg('response ready. sending...')
        payload = "Three rings for the elven kings under the sky"
        response = coap.Message(code=coap.CONTENT, payload=payload)
        d.callback(response)

class TimeResource(resource.CoAPResource):
    def __init__(self):
        resource.CoAPResource.__init__(self)
        self.visible = True
        self.observable = True

        self.notify()

    def notify(self):
        log.msg('TimeResource: trying to send notifications')
        self.updatedState()
        reactor.callLater(60, self.notify)

    def render_GET(self, request):
        response = coap.Message(code=coap.CONTENT, \
```

```
            payload=datetime.datetime.now().strftime("%Y-%m-%d %H:%M"))
        return defer.succeed(response)

class CoreResource(resource.CoAPResource):
    """
    Example Resource that provides list of links hosted by a server.
    Normally it should be hosted at /.well-known/core
    Resource should be initialized with "root" resource, which can be used
    to generate the list of links.
    For the response, an option "Content-Format" is set to value 40,
    meaning "application/link-format". Without it most clients won't
    be able to automatically interpret the link format.
    Notice that self.visible is not set - that means that resource won't
    be listed in the link format it hosts.
    """

    def __init__(self, root):
        resource.CoAPResource.__init__(self)
        self.root = root

    def render_GET(self, request):
        data = []
        self.root.generateResourceList(data, "")
        payload = ",".join(data)
        print payload
        response = coap.Message(code=coap.CONTENT, payload=payload)
        response.opt.content_format = coap.media_types_rev['application/link-format']
        return defer.succeed(response)

# Resource tree creation
log.startLogging(sys.stdout)
root = resource.CoAPResource()

well_known = resource.CoAPResource()
root.putChild('.well-known', well_known)
core = CoreResource(root)
well_known.putChild('core', core)

counter = CounterResource(5000)
root.putChild('counter', counter)

time = TimeResource()
root.putChild('time', time)

other = resource.CoAPResource()
root.putChild('other', other)

block = BlockResource()
```

```
other.putChild('block', block)

separate = SeparateLargeResource()
other.putChild('separate', separate)

endpoint = resource.Endpoint(root)
reactor.listenUDP(coap.COAP_PORT, coap.Coap(endpoint)) #, interface="::")
reactor.run()
```

CoAP 其实是对应于 HTTP 的网络资源访问协议，它是以网络资源为对象的。txThings 基于 Twisted，很自然地，其网络资源的访问方法继承了 Twisted Web 的方法，但是这种方式比较原始。HTTP/CoAP 的路径是类似的：

```
http://<domain>/orgnization/project/device/datapoint/20160301
coap://<domain>/temperautre
```

以上这种 URL 很常见。但是 Twisted Web 的 URL 设计需要逐级访问各个资源。每一级资源对应一个类和代码，显得特别烦琐。所以，在 Twisted Web 基础上，Klein Web 框架增设了 Werkzeug 包。Werkzeug（德语：工具）组件是符合 WSGI 规范的实用函数库，许多 Web 框架如 Klein 和 Flask 都采用 Werkzeug 作为基础函数库。使用 Werkzeug URL 路由装饰器就可以一次定位并访问到目标资源。

txThings 开发者答应笔者会增加其 URL 路由功能。所以，txThings 应该是一个值得期待的软件包。毕竟 CoAP 与 MQTT 相比，其关注度不够高，但更加适合 MCU 的应用场合，其使用场合非常多。而且这种 ROA（Resource Oriented Architect）的方式也可以简化物联网接入设计的逻辑。

前面提到 MQTT 可以不经数据库就直接转发给其余客户端，这是由 MQTT 内置消息队列实现的。CoAP 也可以通过 MultiCast 组播在局域网内实现直接转发。但是两者在适用范围和实现方法上有所不同。CoAP 更加适合无线传感器网络的网内组播，所以采用 CoAP 的设计主要有 TinyOS 和 6LowPAN 等工程。

除了类似于 HTTP，CoAP 还隐含了另外一种设计：设备端实现服务器。在嵌入式设备中实现 CoAP 服务器，而云端计算机集群作为客户端来访问设备端内的服务器。另外，CoAP 设备端服务器可以在数据变化后异步推送给云计算客户端。这减少了不必要的数据流动。其实现可以参考 txThings 的 client_get/client_put 等客户端实现代码。由于 UDP 网络始终在内网穿透上存在弱点，因此 IETF 开始规划基于 TCP 的 CoAP 协议。读者可以持续观察其进展。

9.4.5 MQTT

MQTT（Message Queue Telemetry Transport，消息队列遥测传输）作为一种轻量级的基于代理（broker）的发布/订阅（Publish/Subscribe）消息协议，多年前由 IBM 推出，最近随着物联网

的火热而被许多开发者所熟悉。该协议设计的目的是开放、简单、轻量、容易实施。这些特征使得 MQTT 非常适合资源受限环境，其适用于以下应用场景：

- 网络传输成本较高，带宽较窄，传输不可靠；
- 处理器和存储资源有限的嵌入式设备；
- 设备间、客户端间、服务器间的数据分享。

9.4.5.1　MQTT 协议功能

MQTT 协议功能如下：

- 发布/订阅消息模式提供一对多关系的消息转发和应用间解耦。
- 消息传输与负荷内容无关，底层无须知道高层内容，这对于分层设计有利。
- 利用 TCP/IP 提供基本网络连接能力。
- 提供三个等级的消息传递服务质量（QoS）：最多一次、最少一次、恰好一次。
- 传输消耗很少，其固定长度报文头只有 2 字节，协议交换最小化，可以减少网络流量。
- 实现对于离线客户端的通知方式，以及遗嘱（Last Will）和证明（Testament）功能。

9.4.5.2　三种 QoS 应用场景

由于底层 TCP/IP 网络的原因，同样可能出现报文丢失或者重复，因此必须要约定不同的 QoS。

- 最多一次：适用于环境传感器数据采集。因为即使丢失某些数据，也会马上被新数据填补。
- 最少一次：确保报文一定传输到，由上层应用负责过滤重复发送的报文。
- 恰好一次：适用于金融支付，必须传输到对方，而且只传输一次，否则容易产生多扣款等情况。

表 9-7 列出了 MQTT 协议中使用的消息（message）。

表 9-7　MQTT 协议消息

助记符	枚　举	描　述
Reserved	0	保留
CONNECT	1	客户端连接请求
CONNACK	2	连接确认
PUBLISH	3	发布消息（Publish message）
PUBACK	4	发布确认（Publish Acknowledgment）
PUBREC	5	发布接收，Publish Received （assured delivery part 1）
PUBREL	6	发布发行，Publish Release （assured delivery part 2）
PUBCOMP	7	发布结束，Publish Complete （assured delivery part 3）

续表

助记符	枚　举	描　述
SUBSCRIBE	8	客户端订阅（Subscribe）请求
SUBACK	9	订阅确认
UNSUBSCRIBE	10	客户端去订阅（Unsubscribe）请求
UNSUBACK	11	去订阅确认
PINGREQ	12	PING 请求
PINGRESP	13	PING 回复
DISCONNECT	14	客户端断开
Reserved	15	保留

MQTT 与 REST 方式的最大区别在于数据可以不经过数据库服务器而直接转发。之前从设备到用户端的 CoAP/HTTP/REST 的数据流向如下：

设备端→代理服务器→Web 服务器→SQL/NoSQL 数据库服务器→Web 服务器→代理服务器→客户端

而 MQTT 的数据流向如下：

设备端→MQTT broker→SQL/NoSQL 服务器
设备端→MQTT broker→客户端

所以，MQTT 在设备间流转不需要流经数据库，其实时性很强。但是如果需要保存所有的数据采集，则需要 MQTT 客户端负责将数据录入数据库中。这种模式与传统模式存在较大差别。

实际上物联网的应用之间存在较大差异：一些应用以数据采集为主，入网数据量大于出网数据量；而另一些应用以对等通信为主，入网和出网数据量差不多。大家需要有针对性地增减 MQTT 客户端数量，以满足数据入库不丢失数据为准。

9.4.5.3 套接字服务器升级

在传统物联网应用中，大量采用基于 socket 设计套接字服务器，并构建私有协议。而在物联网协议中，CoAP 基于 UDP 协议，MQTT 基于 TCP 长连接，其通信基础类似，差异仅在于通信协议。但是从工程实施角度来看，从 socket 套接字升级到 CoAP/MQTT 还需要做不少工作：

- 将私有协议报文结构和协议升级到 MQTT。
- 从 IP 地址+端口方式升级到域名+端口方式，增加 DNS 解析。
- 增加 TLS/D-TLS 安全证书相关设计。
- 选配安装 HTTP/HTTPS 服务器，原因有许多。比如 CoAP 作为 UDP 之上的 REST，可以与基于 Web 的 REST 一一对应；MQTT 需要密钥鉴权等。

MQTT 的生产环境部署除了解决技术问题外，还需要先解决一些商务问题。

MQTT 也可以采用 IP 地址，不配置 TLS 安全证书。这降低了使用门槛，但是却给系统带来了安全隐患。而要配置 TLS 安全证书，首先要配置域名，然后根据域名配置 TLS 证书，因此设备端的使用门槛也被提高了。虽然基于 IP 地址可以采用自己颁发证书的方式，但是笔者不推荐这种方式。此外，MQTT 鉴权的密钥需要先从 HTTP/REST 获取，这也对 HTTP/HTTPS 的密钥存在依赖。某些云计算供应商的 MQTT 还规定参与数据分享的服务器必须也是同一供应商的客户。所有这一切，都需要调研清楚，否则需要自行采用开源 MQTT 组件进行定制设计。

此外，大部分支持 MQTT 的设备都是 Linux SBC 和 ARM M3/M4 MCU。对于许多资源有限的嵌入设备，实现 MQTT+TLS 是有难度的，其大部分通过网关汇集数据之后，才通过网关中的 MQTT 客户端推送到 MQTT 服务器。

9.4.6　mosquitto/paho

开源的 MQTT broker 目前最火的开源工程是 mosquitto（蚊子）。该项目采用 C/C++编写，提供各种语言的绑定，包括 Python 和 Node.js。mosquitto 的开发者将其 Python 客户端捐献给了 Eclipse 基金会，并重命名为 Eclipse paho。

9.4.6.1　Linux

在 Linux 中可以对 mosquitto 进行评估。根据其官网文档，其在 Ubuntu 中的安装流程如下：

```
sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
sudo apt-get update
sudo apt-get install mosquitto mosquitto-clients libmosquitto0-dev libmosquittopp0
sudo pip install paho-mqtt
```

Ubuntu 的具体软件包名称，尤其是 lib 文件，可能因版本不同而有所不同，请查看本章延伸阅读部分。Ubuntu 虽然是频繁更新的 Linux 发行版，但是 MQTT 的更新更快，有可能会遇到 MQTT broker 和 client 支持的 MQTT 协议版本不一致的问题。这时，需要根据 mosquitto 官方文档所述进行升级或者源码安装。安装后的路径如下：

```
/usr/sbin/mosquitto
/usr/bin/mosquitto*
/etc/mosquitto/*
/etc/init/mosquitto.conf
```

9.4.6.2　Windows

在 Windows 中可以到 mosquitto 官网下载安装文件，但是该文件还需要用户自行去另外两个网站分别下载其对应的依赖软件包 Win32OpenSSL 和 pthreadVC2.dll。

9.4.6.3 部署

mosquitto 采用 C 语言编写，也采用了异步 I/O 编写，单机可以支持 2 万个长连接。这对大多数中小规模的物联网应用已经足够了。如果超过 2 万个连接，则因为 mosquitto 不支持集群部署，所以需要更多的连接，需要另外选择技术方案。读者可以自己修改源码实现集群，或者使用云计算供应商提供的 IoT PaaS 服务。

启动：

```
sudo service mosquitto start
```

停止：

```
sudo service mosquitto stop
```

查看状态：

```
root@iZ2573cw0yvZ:~# sudo service mosquitto status
mosquitto start/running, process 24835

root@iZ2573cw0yvZ:~# netstat -anplt | grep 1883
tcp        0      0 0.0.0.0:1883          0.0.0.0:*         LISTEN      24835/mosquitto
tcp6       0      0 :::1883               :::*              LISTEN      24835/mosquitto
```

从 netstat 可以看出，mosquitto 还是 IPv6 兼容的。可以使用以下指令实现对于 MQTT broker 的发布和订阅：

```
mosquitto_sub
mosquitto_pub
```

测试 mosquitto 公网测试信息，可以订阅并读取大量的传感器数据，在此可以发现各种数据类型：JSON、TEXT、Base64。

```
sudo mosquitto_sub -h test.mosquitto.org -t "#" -v
```

9.4.6.4 测试

在 Ubuntu 中运行一个 mosquitto broker。在其中一个终端中输入：

```
mosquitto_sub -h 192.168.1.107 -t /demo/sensor
```

即在 192.168.1.107 主机（域名也可以）订阅/demo/sensor 主题的消息。

在另外一种终端中输入：

```
mosquitto_pub -h 192.168.1.107 -t /demo/sensor -m 'Hello world'
```

即向 192.168.1.107 主机/demo/sensor 主题中推送发布消息'Hello world'。

在 MQTT 中，broker、subscriber、publisher 构成三个角色，但 subscriber 和 publisher 可以

是多个独立实体，以构成 N：M 关系。

paho-publisher.py：

```
import paho.mqtt.client as mqtt
import random
import time

def on_connect(client, userdata, rc):
    print("Connected with result code "+str(rc))
    client.publish("/allankliu/Devices", str(random.randint(0,9999)))

def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect('test.mosquitto.org', 1883, 60)

i=0
while 1:
    client.publish("/EPIC/Devices", 'flow='+str(random.randint(0,9999)))
    i+=1
    print("MQTT pub %d"%(i))
    time.sleep(0.5)
```

paho-subscriber.py：

```
import paho.mqtt.client as mqtt

def on_connect(client, userdata, rc):
    print("Connected with result code "+str(rc))
    client.subscribe("/allankliu/#")

def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("test.mosquitto.org", 1883, 60)

client.loop_forever()
```

笔者在阿里云中构建 mosquitto broker，测试通过。

9.4.6.5 twisted-mqtt-client

Eclipse paho 适合作为设备域中的 MQTT 客户端。在服务器域中，前面提到的 Twisted mosquitto MQTT 客户端也已经在 GitHub 上出现。这意味着在服务器上安装 Twisted 和 mosquitto 之后，利用该客户端就可以构建一个完整的 MQTT 物联网方案。

Twisted mosquitto MQTT 客户端依然是单线程异步架构。如果接入 MQTT 的设备并发连接数很高，则需要在 MQTT broker 后并发连接多个 Twisted MQTT 客户端，这些客户端再连接数据库。这种 MQTT broker 前置，后端有多个工作客户端的模式，很类似于 Ngnix + Web Server 负载均衡的设计。我们来看一下参考代码。

twisted_mqtt_client.py：

```
import sys

from twisted.internet import reactor, task
from twisted.application.internet import ClientService
from twisted.internet.endpoints   import clientFromString
from twisted.logger   import (
    Logger, LogLevel, globalLogBeginner, textFileLogObserver,
    FilteringLogObserver, LogLevelFilterPredicate)

from mqtt.client.factory import MQTTFactory

# ----------------
# Global variables
# ----------------

# Global object to control globally namespace logging
logLevelFilterPredicate = LogLevelFilterPredicate(defaultLogLevel=LogLevel.info)

# -----------------
# Utility Functions
# -----------------

def startLogging(console=True, filepath=None):
    '''
    Starts the global Twisted logger subsystem with maybe
    stdout and/or a file specified in the config file
    '''
    global logLevelFilterPredicate

    observers = []
    if console:
        observers.append( FilteringLogObserver(observer=\
            textFileLogObserver(sys.stdout),\
            predicates=[logLevelFilterPredicate] ))
```

```
    if filepath is not None and filepath != "":
        observers.append( FilteringLogObserver(observer=\
            textFileLogObserver(open(filepath,'a')),\
            predicates=[logLevelFilterPredicate] ))
    globalLogBeginner.beginLoggingTo(observers)


def setLogLevel(namespace=None, levelStr='info'):
    '''
    Set a new log level for a given namespace
    LevelStr is: 'critical', 'error', 'warn', 'info', 'debug'
    '''
    level = LogLevel.levelWithName(levelStr)
    logLevelFilterPredicate.setLogLevelForNamespace(namespace=namespace, \
            level=level)


class MyService(ClientService):

    def gotProtocol(self, p):
        self.protocol = p
        d = p.connect("TwistedMQTT-pubsubs", keepalive=0)
        d.addCallback(self.subscribe)
        d.addCallback(self.prepareToPublish)

    def subscribe(self, *args):
        d = self.protocol.subscribe("foo/bar/baz", 0 )
        self.protocol.setPublishHandler(self.onPublish)

    def onPublish(self, topic, payload, qos, dup, retain, msgId):
       log.debug("msg={payload}", payload=payload)

    def prepareToPublish(self, *args):
        self.task = task.LoopingCall(self.publish)
        self.task.start(5.0)

    def publish(self):
        d = self.protocol.publish(topic="foo/bar/baz", message="hello friends")
        d.addErrback(self.printError)

    def printError(self, *args):
        log.debug("args={args!s}", args=args)
        reactor.stop()


if __name__ == '__main__':
    import sys
```

```
log = Logger()
startLogging()
setLogLevel(namespace='mqtt',     levelStr='debug')
setLogLevel(namespace='__main__', levelStr='debug')

factory    = MQTTFactory(profile=MQTTFactory.PUBLISHER | MQTTFactory.SUBSCRIBER)
myEndpoint = clientFromString(reactor, "tcp:test.mosquitto.org:1883")
serv       = MyService(myEndpoint, factory)
serv.whenConnected().addCallback(serv.gotProtocol)
serv.startService()
reactor.run()
```

在 MQTT/Ngnix 负载均衡设计中，mosquitto MQTT 单核可以支持 2 万个长连接，Ngnix 单核可以支持 50 万个连接。而下一级工作服务器单台实例成本并不高。以单台负载均衡配合 8 台工作服务器计算，总成本并不算高。所以，笔者对于这种架构设计很有信心，其可以满足大部分中小企业的物联网接入需求。假设每台设备（如手环）的销售价格为 100 元，则 2 万到 5 万个单品的销售额在 200 万元到 500 万元间。如果同时运行多种设备，则年销售额在千万元级别。资产远远超过云端服务器的成本。

9.4.7 REST API

REST API 已经成为网络应用的标准配置。严格地说，基于 JSON 的 REST API 比原先基于 XML 的 Web Service 接口要缺少一些特性。但是，轻量化的 JSON 渐渐替代了 XML 方式。API 的设计和使用需要满足认证、鉴权、计费，兼顾设计复杂度和使用方便性，可以参考开源设计进行实现。一些云计算服务商提供了 API 网关，可以简化 API 开发。

9.4.8 服务器数据推送技术

本节介绍的服务器数据推送技术大部分基于浏览器/服务器模型，但是物联网应用可以借鉴使用某些技术，比如 WebSocket。

在传统 Web 服务器设计中，服务器持续侦听特定端口的数据。因为服务器在连接前无法知晓客户端的 IP 地址和端口，所以启动通信会话的都是客户端。客户端可以从服务器提取数据（如 HTTP GET），也可以向服务器写入数据（如 HTTP POST），数据流向是双向的。HTTP GET/POST 都是短连接，获取新数据需要重发请求。

许多应用需要周期性持续获得服务器的信息：如实时绘图或数据长时间转发分享。最基本的实现方式是利用 JavaScript XMLHttpRequest 方法每隔几秒从服务器数据端口 URL 读取一次数据并更新绘图区域。至于跨域问题，可以使用 JSONP（JSON with Padding）跨域返回 JavaScript 代码，并内嵌 JSON 数据来解决。这种基于 HTTP/AJAX 短连接的方式被称为轮询（polling）。

HTTP 请求头和响应头比较长，而负荷数据可能并不多。甚至在许多物联网应用中，服务器端中保存的物理量数据并没有更新，而客户端为了及时更新必须周期性地轮询，这耗费了大量带宽。所以，许多实现方案转向长连接和服务器主动推送。

在长连接中，由服务器持续或者周期性地主动向客户端或浏览器发送数据，可以减少 HTTP 请求头的带宽使用。针对服务器推送数据的解决方案尤其多，其大多数是针对服务器和浏览器间的数据推送。主要方案如下：

- Java RMI（Remote Method Revoke），需要浏览器安装插件。
- Java CORBA（Common Object Request Broker Architecture），需要浏览器安装插件。
- Java Applet，需要安装 Java 插件。
- Flash XMLSocket，Adobe Flash + JavaScript 组合，需要安装 Flash。
- 反向 AJAX，AJAX 只可以由客户端发起通信。但通过模拟客户端和服务器通信，可以促使服务器主动推送消息给客户端。
- piggyback polling（背包轮询），请求响应分为两部分：对于本地请求的响应和上一次请求的响应。
- 基于 HTTP 长连接，无须插件的技术被称为 Comet。Comet 有两种实现：长轮询（long-polling）和流（Streaming）方式。

9.4.8.1 long-polling

传统 AJAX 使用 JavaScript 的 XMLHttpRequest 方法向服务器发出请求，并在服务器返回信息后响应并修改 HTML 局部内容。但 AJAX long-polling 与传统 AJAX 的不同点如下：

- 服务器会阻塞客户端请求，直至有数据传递或者超时才返回。
- 客户端 JavaScript 响应处理函数在处理完信息后，需要重新发出请求，建立连接。
- 客户端处理数据或重连时，服务器端有可能有新数据到达。这些数据会被服务器保存或缓存，直至客户端重新建立连接后一次性发送给客户端。

9.4.8.2 Streaming

iframe 是一种很古老的 HTML 标记，Streaming 利用隐藏 iframe 实现长连接。通过一个隐藏的 iframe，并将 iframe src 设定为某个长连接 API 请求，服务器可以源源不断地向客户端输入数据。其缺点是浏览器加载进度圈会不停地转。为此，Google 提供了一个 htmlfile 的 ActiveX 空间解决这个问题。

XMLHttpRequest.readystate 的 0~4 所对应的状态如下：未初始化、载入、载入完成、交互与完成。与 long-polling 和 polling 不同的是，Streaming AJAX 在 XMLHttpRequest.readystate == 3 即载入完成时，就可以读取数据，而且无须关闭连接。

9.4.8.3 multi-part

XMLHttpRequest 对象上使用某些浏览器（比如说 Firefox）支持的 multi-part 标志。AJAX 请求被发送给服务器端并保持打开状态，每次有事件到来时，一个（multi-part）多部件响应就会通过同一连接来推送给客户端。

9.4.8.4 SSE/WebSocket

以上这些大多数都与浏览器相关，涉及 JavaScript 编程。HTML5 为服务器推送提供了另外两种技术实现：SSE 和 WebSocket。

- SSE，Server Sent Event，服务器发送事件。
- WebSocket，实现了浏览器和服务器全双工通信，但握手阶段需要借助 HTTP 完成。

SSE 还是基于 HTTP 访问，所以它的 URI 与 HTTP 一样：

- http://yourweb:port/path；
- https://yourip:port/path （带安全证书）。

WebSocket 作为一种全新的协议，其 URI 为：

- ws://yourip:port/path；
- wss://yourip:port/path （带安全证书）。

这两者的区别如下。

- WebSocket 全双工通信，SSE 由服务器单向推送给客户端或浏览器。
- WebSocket 作为全新协议，需要服务器端升级；而 SSE 部署在 HTTP 之上，修改较小。
- SSE 较为轻量、简单。
- SSE 支持断线重连，WebSocket 需要额外设计。
- SSE 主要采用文本协议，但是也可以自定义协议。
- WebSocket 适合大量二进制数据的传输。
- WebSocket 虽然有不少实现，但是实现之间的区别较大。

对于 SSE/WebSocket，Python Web 框架中基本都增加了对应扩展，如：Tornado-SSE、Cyclone-SSE/Cyclone-WebSocket 或 PyWebSocket 等。现阶段大多数的服务器推送的对象主要是浏览器，但实际上也可以是任何 APP 或者第三方服务器。

在实际工程中，物联网应用系统架构都是端、管、云。数据从设备流向中心服务器。虽然服务器分享可以采用 REST API 进行，但是由于大多数服务器都是被动侦听模式，因此如果没有 MQTT 代理服务器或者消息队列，服务器之间彼此就无法主动推送数据。这时，需要其中一台服务器主动将数据推送给其他服务器。这时候可以利用 SSE 进行数据推送。

9.5　高可用性与高并发性

“高可用性”（High Availability）通常用来描述一个系统经过专门的设计，从而减少停工时间，而保持其服务的高度可用性。其主要手段有冗余备份、负载均衡和集群。

“高并发性”（Concurrency），主要指服务器系统可以同时接入大量甚至是巨量的客户端。其实现手段包括采用面向并发性能的语言和框架、服务器架构优化、负载均衡和分布式计算。

无论是物联网还是互联网，数据量和连接数都会逐渐增加，其服务器架构需要针对高可用性、高并发性进行设计。

9.5.1　并行与并发计算

Go 语言的开发者 Rob Pike 及 Real world Haskell 的作者提到一点：并发（Concurrency）与并行（Parallelism）是有区别的：

- 并发程序指能同时执行通常不相关的各种任务，其并不需要多核处理器。
- 并行程序是用来解决一个单一任务的，基于硬件的冗余可以实现并行计算。

以负载均衡+多台服务器的使用场景为例，Ngnix 反向代理服务器为并发计算，后端多台工作服务器是并行计算。

新的计算核心和计算模式如超标量、超流水线、超线程、超长指令集、多核计算、网格计算都能够在某种程度上增加并行计算数量。能够实现的并行计算指令是核心数量乘以架构允许的并发数量。

- 以 Intel Core 笔记本 i3 CPU 为例：具备 2 核心、4 线程，其可以支持 8 个指令并行计算。
- 以 NVIDIA Tesla Kepler GK210 GPU 为例，其具备 2496 个计算内核。
- 以 NVIDIA GeForce GTX860M 为例，其具备 640/1152 个计算内核。
- NVIDIA/AWS/阿里云的 HPC 集群，可以让客户充分利用 GPU 计算集群用于科学计算。

选择一种网络编程框架，需要先了解该框架使用了何种底层技术和库。实现并发的模型主有 I/O 复用、多进程、多线程和协程。这些模型都需要依赖于操作系统提供的系统调用，同时在编程语言、网络编程框架中对相关模型进行支持。前面已经介绍了 Python 语言对于多进程、多线程和协程的支持，接下来介绍 I/O 模型与分类。

9.5.2　网络 I/O 模型分类

不同操作系统中的 I/O 模型如下：

- 阻塞型 I/O：用户进程一直处于等待状态，直到 I/O 复制操作完毕。
- 非阻塞型 I/O：内核数据没有就绪时立刻返回，进程可以处理其他任务，并周期性轮询；

数据就绪后，全程参与数据复制过程，直至复制完毕。

- I/O 复用：进程在等待数据和复制数据阶段均处于等待状态，整个过程有两次系统调用，但是系统会同时监控多个端口，以实现等待状态时的处理时间复用。
- 信号驱动 I/O：进程在等待数据时采用回调方式，复制数据阶段参与复制。
- 异步 I/O：用户进程发起 I/O 操作后，可以处理其他任务，直到内核通知数据就绪。

I/O 复用让应用程序可以同时监控多个 I/O 操作，通过切换上下文实现对于某个时间数据就绪的 I/O 进行控制。Linux/UNIX 中的 seletc/poll/epoll/kqueue 都是 I/O 复用，而 SIGIO 归属于信号驱动，Windows IOCP 属于异步 I/O 模型。许多事件驱动库如 libev/libuv/libevent 均封装了若干种 I/O 模型，并提供给网络编程框架如 Node.js/Twisted 使用。

9.5.3 架构优化的路径

无论是互联网还是物联网，其网站架构可能有所不同，但是其架构优化的目的和策略是一致的，即采用各种方式针对数据流动和性能瓶颈进行优化，实现弹性动态扩展和数据流动的畅通无阻。主要方式如下：负载均衡、数据缓存、时序数据库与消息队列。

单一实例的服务器的并发数是有限的，所以采用负载均衡的方式将所有的流量和处理任务分摊到多台服务器，在互联网行业中是一件较为普遍的做法。负载均衡不仅仅在设备接入端节点使用，还在服务器集群内部用于化解各类数据瓶颈，比如数据库的负载均衡等。

数据缓存已经在许多互联网应用中被采用。由于数据缓存利用 RAM 作为主要的存储媒体，因此其存储数据非常快。转发给其他服务器也非常快。当然，缓存的容量是有限的，所以，其只适合保存一定规模的“热点”数据。这在物联网数据采集系统中应用得非常广泛。

时序数据库完全根据物联网的特点而定制。由于大多数时序数据库还比较新，因此目前还没有形成主流产品。但是使用时序数据库建模比基于 SQL/NoSQL 的建模要简单得多。检索、删除、统计也更加简单、快捷。

消息队列可以将紧耦合服务器拆分成多个异步事件处理子系统，彼此通过消息队列形成松耦合系统，这可以相对简化系统设计和运维成本。

当系统规模比较小时，可以采用各种开源组件降低启动成本。同时，云计算供应商都已经提供了大量对应的云服务组件，用户基于这些组件可以大大降低运维成本，并实现系统规模的无缝扩展。

9.5.4 关系数据库系统

RDBS（Relational Data Base System，关系数据库）是最常用的数据库，适用于二维表格类型数据。应用程序采用 SQL（Structured Query Language）语法进行数据管理。

MySQL 是瑞典 MySQL AB 公司的产品，后被 Sun 公司收购，现在其为 Oracle 旗下产品。MySQL 是 LAMP（Linux+Apache/Ngnix+MySQL+PHP/Perl/Python）“四大金刚”之一，它曾经是笔者常用的一个开源组件。

MariaDB 是 MySQL 的一个分支，由 MySQL 的创始人 Michael Widenius 主导开发。MariaDB 是 MySQL 的二进制替代品。其采用 XtraDB 替代了 MySQL 的 InnoDB 存储引擎，使用 Maria 存储引擎替换了 MySQL 的 MyISAM 存储引擎。由于 MySQL/Sun 被 Oracle 收购后有闭源的风险，因此 MariaDB 更具吸引力。MariaDB 在功能扩展、存储引擎以及新特性方面的发展都非常快。

另外一种常见的 RDBS 是 PostgreSQL，其前身是美国加州大学伯克利分校计算机系开发的 Postgres，它是一种对象关系型数据库管理系统（ORDBMS）。用户可以免费使用、修改和分发 PostgreSQL。PostgreSQL 比 MySQL/MariaDB 更加复杂，其适合企业级应用。

PostgreSQL 支持大部分 SQL 标准并且提供了许多其他现代特性：复杂查询、外键、触发器、视图、事务完整性、MVCC。同时，PostgreSQL 可以用许多方法扩展，比如增加新的数据类型、函数、操作符、聚集函数、索引方法和过程语言。

ORM

在编程语言中，可以通过 SQL 语法直接操作 RDBS，但拼合查询字符串是最常见的网络攻击原因之一，也非常容易出错。我们在软件开发中往往希望能够将编程语言中的对象直接映射到数据库表中。ORM（Object Relational Mapping，对象关系映射），作为一种程序技术，用于实现面对对象编程语言中不同类型系统的数据间转换。实际上，ORM 创建了一种编程语言中的“虚拟对象数据库”。

虽然大部分关系数据库都采用 SQL 作为接口，但是不同数据库之间还是存在些许差异的。Python ORM 技术可以帮助我们简化编程难度，可以不用操作底层的 SQL 字符串，而直接存取 Python 对象。

1）ORM 的优点

- 简单：ORM 以基本形式建模。将表映射为类，表字段就是类成员变量。
- 精确：应用系统不再依赖于特定数据库语法，并在代码层面保持准确、统一。
- 易懂：ORM 使得数据库结构文档化。
- 易用：ORM 避免了不规范、冗余、风格不统一的 SQL 语句，避免了人为的软件 bug。
- 扩展：ORM 类是可以继承的。

ORM 对于敏捷开发和团队合作的好处非常大。

2）ORM 的缺点

- 性能：额外的自动化映射耗费一定的系统资源。
- 复杂查询：处理多表联查或复杂 where 条件查询时，查询也很复杂。

- 消耗内存：其比 SQL 直接查询消耗内存。

所以，在一些大数据量、大运算量和复杂查询时，可以视情况在 ORM 中混用 SQL 查询。

3）Python ORM

Python ORM 有很多种类，不同的 ORM 有不同的学习周期和内部设计。

- SQLAlchemy 是一个非常流行的 Python ORM，是 Python ORM 的事实标准；在许多框架中都使用了 SQLAlchemy。
- SQLObject 是另外一个优秀的 Python ORM，类似于 Ruby on Rails 的 ActiveRecord，其现在日渐流行。
- PyDO 是 Python 对象映射库，支持 PostgreSQL、SQLite、MySQL、MSSQL 和 Oracle。

此外还有 Storm、Django ORM、peewee 等 ORM，选择非常多。推荐大家从最主流的 SQLAlchemy 开始学习。

9.5.5 SQL/NoSQL/NewSQL

传统数据库是结构化的关系数据库（RDBS）。结构化查询语言 SQL 是 RDBS 的标准接口语言。然而随着大规模和高并发的 Web 2.0 社交应用的出现，传统 SQL 数据库变得无法适应。于是。各种 NoSQL 得到了迅速的发展。NoSQL（Not only SQL）数据库的产生是为了应对大规模数据集合、多种数据种类的挑战，尤其是大数据应用的难题。表 9-8 中列出了常见的 NoSQL 数据库。

表 9-8　NoSQL 数据库分类

分　　类	典型例子	数据模型	应用场景	优　　点	缺　　点
键值型	Redis	哈希表	内容缓存，大量数据热点负荷，日志系统	查找速度快	数据非结构化
列存储型	Cassandra，HBase	列簇存储，一键多列，指向分布式服务器中的多个列对象，同一列数据存在一起	分布式文件系统	查找速度快，可扩展性强，容易分布式存储	与 RDBS 相比，功能简单
文档型	MongoDB	键值对，数值部分为 JSON 半结构化数据	与键值类似，但数据库可以了解值的内容	数据结构要求不高，无须事先定义表结构，允许数据冗余	查询性能不高，缺乏统一语法
图形	Neo4J	图结构，可扩展到多个数据库	社交网络，推荐系统，关系图谱	最短路径寻址，*N* 度关系查找	需遍历图结构，不太好做分布式集群

NoSQL 数据库在以下情况下比较适用：

- 数据模型比较简单；
- 系统灵活性更强；
- 数据库性能要求较高；
- 无须高度的数据一致性；
- 单一键更容易映射复杂数值。

NoSQL 无法完全替代 RDBS，目前还面临许多挑战：

- 键值数据库产品缺乏通用性，多面向特定应用构建；
- 产品支持功能有限，往往不支持事务特性，导致应用受限制；
- 缺乏 RDBS 所具有的理论、技术和标准规范；
- 缺乏安全的重要设计；
- 新型应用不仅仅要求可扩展性、弹性、容错，还需要自管理和强一致性。

在此基础上，出现了 NewSQL。此类数据不仅具备 NoSQL 对于海量数据的存储管理能力，还保持了传统数据库的 ACID 和 SQL 特性。此类 NewSQL 包括 Clustrix、GenieDB、ScaleBase、带有 NDB 的 MySQL 集群和带有 HandlerSocket 的 MySQL，还包括 Amazon DB、SQLAzure、Xeround 和 FathomDB。与 NoSQL 相比，NewSQL 主要满足了结构化、支持 SQL 语法，以及可扩展性和查询特性。但开发者使用前需要充分了解相关技术，以适应自身应用需求。

9.5.6　Redis

Redis 是 NoSQL 中比较著名的一款键值型内存数据库。它和 Memcached 有一些场景是重叠的，但 Redis 覆盖的场景比 Memcached 要多，其支持的数据类型和数据结构也更多。笔者更加喜欢将 Redis 用在自己的设计中。

Redis 采用 ANSI C 编写，作者为 Salvatore Sanfilippo。Redis 支持网络操作，是一种基于内存的键值型 NoSQL 数据库，同时也支持数据库持久化，并提供各种语言接口。

Redis 提供了五种数据类型：string、hash、list、set 及 zset（sorted set）。这些数据类型各有特点以及限制，适用于不同场景，需要开发者仔细考虑、挑选。

Redis 的缺陷是持久层方面完全不是为了数据存储而设计，而是为了保存 Redis 运行状态而设计的。换而言之，Redis 的持久层设计是 Redis 运行时持久，而非 Redis 数据持久。

在笔者开发的 EPIC 系统中，将 Redis 用于以下场景：

- 利用 string 类型保存实时热点数据缓存，并利用超时自动删除特性过滤数据；
- 利用 hash 类型保存连接状态（IP 地址、端口、传输数据统计等）；
- 利用 zset 类型保存地理位置（国家、城市排序）；
- 利用 Redis Pub/Sub 模式，部分替代消息队列。如果系统中已经使用了 Redis，则可以

充分利用 Redis 的这一特性来实现不同服务器之间的进程间异步通信。

在使用过程中，笔者发现 Redis 有一个关机 bug。因为 Redis 默认没有激活密码，所以它的关机脚本也没有配置密码，无法正常关闭 Redis，以致无法关机。所以，笔者增加了一个脚本，通过 redis-cli 方式以密码方式关闭 Redis 守护进程，然后再关机。

redis_shutdown.sh：

```
#!/bin/bash
redis-cli -a "redis-password" shutdown
#shutdown -P now
```

执行最后一句就可以直接关机，不过需要管理员权限。

9.5.7 MongoDB

MongoDB 是一种文档型数据库，采用 C++编写。其介于 SQL 和 NoSQL 之间，也是 NoSQL 中最像 SQL 的一个产品。MongoDB 可以被视为一种升级的键值数据库。其主要目标是在键值存储以及传统的 RDBMS 系统间架起一座桥梁，集两者的优势于一身。

9.5.7.1 MongoDB 适合的场景

具体场景如下。

- 网站数据：适合实时的插入、更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性。
- 缓存：性能很高，MongoDB 也适合作为信息基础设施的缓存层。在系统重启之后，由 MongoDB 搭建的持久化缓存可以避免下层的数据源过载。
- 大尺寸数据：使用传统的关系数据库存储一些大型数据时成本比较高。很多程序员往往会选择传统的文件系统进行存储，现在可以保存在 MongoDB 中。
- 高伸缩性：MongoDB 非常适合由数十或者数百台服务器组成的分布式数据库。
- 对象存储：可用于对象及 JSON 数据的存储。MongoDB 的 BSON 数据格式非常适合文档格式化的存储及查询。

9.5.7.2 不适用 MongoDB 的场景

具体场景如下。

- 高度事务性数据系统：例如银行或会计系统。传统的关系数据库目前还是更适用于需要大量原子性复杂事务的应用程序。
- 传统商业智能应用：针对特定问题的 BI 数据库需要高度优化的查询方式。对于此类应用，数据仓库可能是更合适的选择。
- 其他需要 SQL 的场景。

笔者曾经非常纠结实时数据是保存在 Redis 还是保存在 MongoDB 中，最终觉得 Redis 的自动删除特性更加适合当时的应用。如果不需要持久保存，那么 Redis 足够在单机中使用；如果这些实时数据需要持久保存，则可以选用 MongoDB。

9.5.8　时序数据库

时序数据库，其全称为时间序列数据库。时间序列数据库主要用于处理带时间标签（按照时间的顺序变化，即时间序列化）的数据，带时间标签的数据也被称为时间序列数据。

时间序列数据主要是由各个行业的各类型实时监测、检查与分析设备所采集、产生的数据，这些工业数据的典型特点如下：

- 产生频率快，每一个监测点的单位时间内可产生多条数据；
- 严重依赖于采集时间，每一条数据均要求对应唯一的时间；
- 测点多、信息量大，常规的实时监测系统均有成千上万的监测点，监测点每秒都产生数据，每天产生几十 GB 的数据量。

基于时间序列数据的特点，关系数据库无法满足对时间序列数据的有效存储与处理，因此迫切需要一种专门针对时间序列数据来做优化的数据库系统，即时间序列数据库。

时序数据库的特点如下：

- 时间流——伴随时间流不断产生数据。
- 多维度——有多个属性，比如各种物理量、经纬度等，需要统计和查询。
- 查询方式——按照时间查询，并计算均值、中值和统计分布。
- 数据量——随着时间的推移，数据体量巨大。
- 响应时间——要求响应时间要快，数据入库即可使用。
- 数据价值——随着时间的推移，数据价值降低。
- 规模扩展——在线水平扩展。

时序数据库的写入特性如下：

- 写多于读——以记录写入为主要方式。
- 顺序写入——数据以实时追加模式写入。
- 极少更新——删除的概率远大于更新。
- 区块删除——会针对一个时间段进行删除。

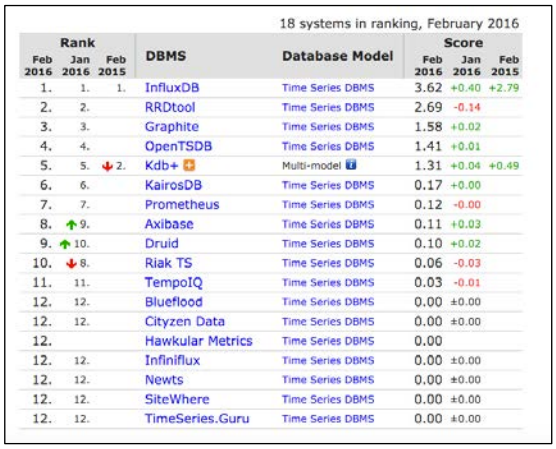
时序数据库的读取查询特性如下：

- 顺序读取——会针对一个时间段进行查询、统计、分析。
- 数据量大——原始数据无法在内存中保存，其大多存储于分布式存储系统中。
- 读取速度快——这是 TSDB 的需求。

时序数据库面临的主要困难如下：数据体量大，维度多，统计查询复杂，需要快速响应查询。时序数据库解决这些问题的主要技术手段如下：

- 分布式处理；
- 时间分片存储；
- 维度条件做哈希；
- 通过位图索引优化查询；
- 最新热点数据保存在内存中；
- 使用统计算法。

DB-engines.com 网站的结果列在图 9-13 中，其中 InfluxDB 是当时最热门的时序数据库。本章延伸阅读部分中有关于如何利用 InfluxDB 和其他组件构建监控系统的内容。因为数据量较大，所以 InfluxDB 生产环境中的版本请选用 64 位版本。



18 systems in ranking, February 2016

Rank Feb 2016	Rank Jan 2016	Rank Feb 2015	DBMS	Database Model	Score Feb 2016	Score Jan 2016	Score Feb 2015
1.	1.	1.	InfluxDB	Time Series DBMS	3.62	+0.40	+2.79
2.	2.		RRDtool	Time Series DBMS	2.69	-0.14	
3.	3.		Graphite	Time Series DBMS	1.58	+0.02	
4.	4.		OpenTSDB	Time Series DBMS	1.41	+0.01	
5.	5.	↓ 2.	Kdb+	Multi-model	1.31	+0.04	+0.49
6.	6.		KairosDB	Time Series DBMS	0.17	+0.00	
7.	7.		Prometheus	Time Series DBMS	0.12	-0.00	
8.	↑ 9.		Axibase	Time Series DBMS	0.11	+0.03	
9.	↑ 10.		Druid	Time Series DBMS	0.10	+0.02	
10.	↓ 8.		Riak TS	Time Series DBMS	0.06	-0.03	
11.	11.		TempoIQ	Time Series DBMS	0.03	-0.01	
12.	12.		Blueflood	Time Series DBMS	0.00	±0.00	
12.	12.		Cityzen Data	Time Series DBMS	0.00	±0.00	
12.			Hawkular Metrics	Time Series DBMS	0.00		
12.	12.		Infiniflux	Time Series DBMS	0.00	±0.00	
12.	12.		Newts	Time Series DBMS	0.00	±0.00	
12.	12.		SiteWhere	Time Series DBMS	0.00	±0.00	
12.	12.		TimeSeries.Guru	Time Series DBMS	0.00	±0.00	

图 9-13　DB-engines.com 的 TSDB 网络调查表

目前可选的时序数据库产品和服务如下：

- InfluxDB，采用 Go 语言编写；
- Hummer TSDB，蜂鸟时序数据库；
- OpenTSDB，基于 HBase，采用 Java 语言编写；
- Cassandra，采用 Java 语言编写；
- Druid，分布式列存储器，采用 Java 语言编写；
- Riak TSDB，采用 Erlang 语言编写，商业版本；
- Geras，采用 Python 语言编写；
- RRDTool（Round Robin Database Tool），采用 C 语言编写；

- Graphite；采用 Python 语言编写；
- KairosDB，基于 Cassandra 存储的 OpenTSDB 分支；
- KDB+，免费版本限制于 32 位；
- SiteWhere，开源 IoT 平台，存储技术采用 MongoDB/HBase；
- TempoIQ，IoT 平台；
- TreasureData，Hadoop 数据处理；
- Baidu TSDB PaaS，百度的时序数据库云计算服务器；
- Bluemix TSDB PaaS，基于 MongoDB；
- PI System，商业版本。

这些 TSDB 都是针对大体量数据而设计的，实施起来可能需要大量计算机资源和很多存储资源。国内百度天工自带的 TSDB 是唯一以时序数据为卖点的时序数据库服务。由于时序数据库目前缺乏主流标准产品，因此需要用户在服务器实例中构建集群。如果在云计算平台中实施，则对系统的 IOPS（Input/ Output Operations Per Second，即每秒读/写次数）有性能要求。

9.5.9　消息队列

消息队列是不同组件间的异步通信利器，可以用于不同的场景。其目前已经成为服务器后端的常见组件之一，主流的消息队列如下：

- ZeroMQ（C++）；
- ActiveMQ（Java）；
- RabbitMQ（Erlang）；
- Kafka（Java）。

9.5.9.1　ZeroMQ

ZeroMQ 即 Zero Message Queue，其缩写为 ZMQ，也可被写成 ØMQ。其类似于 socket 接口，不过不是 1 对 1，而是多对多的接口。ZMQ 用于主机、内核、线程或者进程间异步通信。它使得 socket 编程更加简单、简洁和高效，并提供了 C/C++/Java/.NET/Python 等三十多种开发语言绑定。

ZeroMQ 将消息通信分成四种模型。

- 一对一结对模型：其可以被理解为一个 TCP 连接，数据双向流动。
- 请求/回应模型：由请求端发出请求，等待回应端回应。一个请求对应于一个回应。其可以是 1：N 关系，用于远程调用和任务分配。
- 发布/订阅模型：发布端单向分发数据，适用于天气预报、微博粉丝等场景。
- 推拉模型：其主要用于多任务并行处理。

由于 ZeroMQ 的性能超越一般 MQ，因此在物联网的设备应用中也可以找到它。比如前面提到过的 panStamp SubGHz 网关，就曾经利用 ZeroMQ 来实现 Lagarto-SWAP 和其他组件如 Web 监控页面之间的联系。我们可以在物联网网关设计中充分利用 ZeroMQ 实现内部组件间通信。由于 ZeroMQ 不仅仅是进程间通信，还可用于主机间通信，因此，在 Python 运维组件中 SaltStack 也充分利用了 ZeroMQ 的这一特性实现集群间通信。

由于 ZerMQ、MQTT 和 Redis 三者间在发布/订阅模型应用中存在一定的竞争关系，因此不同侧重点的应用选择结果也会不同。panStamp 在最新的设计中，主要因为 ZeroMQ/MQTT 两者功能重复，而 MQTT 已经成为物联网的标准接入协议，所以采用 MQTT 替代了 ZeroMQ。

9.5.9.2 Celery

异步任务是 Web 开发中一个很常见的方法。对于一些耗时、耗资源的操作，往往需要从主应用中独立出来，通过异步的方式执行。例如，邮件发送是用户注册以及重置密码时常见的方式。如果直接放在应用中，则邮件发送的过程会阻塞网络 I/O，非常耗时。比较优雅的实现方式是使用异步任务，应用在业务逻辑中触发一个异步任务。两者间通过消息队列进行耦合。将电邮任务交给发送电邮进程，发送完毕或失败后，由电邮进程通知主进程。

实现异步任务的工具有很多，其原理都是使用一个任务队列，比如生产消费模型或者发布/订阅模型都是利用消息队列实现的编程模型。

除了 Redis 和各类 MQ 实现多机、多进程和多线程间异步通信外，还可以使用另外一个异步神器：Celery。Celery 是一个异步任务的调度工具。它是 Python 封装库，但是它实现的通信协议也可以使用 Ruby、JavaScript 和 PHP 进行调用。Celery 的异步任务除了实现消息队列的后台执行方式外，还可以实现时间触发的计划任务，以及可以部分替代 crontab。

Celery 的实现独立于存储后端，其 broker 可以是 RabbitMQ、Redis、SQL 数据库以及 Amazon SQS、MongoDB 和 IronMQ。

Celery 进程间通信协议支持 pickle、JSON、msgpack 等各类协议，同时针对各类 Python 框架提供了一些整合工具。很遗憾其还未直接支持 Cyclone/Twisted。不过 Celery 支持 Tornado，而 Cyclone 和 Tornado 共享 API，所以移植工作量不算大。

9.5.9.3 Python-RQ

RQ 是 Redis Queue 的缩写，作者为 Vincent Driessen。RQ 和 Celery 类似，可以管理队列任务，并用 worker 在后台进行处理。RQ 比 Celery 简单，其学习门槛较低，比较适合规模较小的应用。它的 broker 只有 Redis，可以较为容易地和 Web 框架进行整合。

RQ 的限制如下：

- RQ 基于 Redis V2.6.0 或更新的版本。
- RQ 依赖于 fork 方法，所以和 Redis 一样，不支持 Windows 系统。

- RQ 依赖于 pickle，仅限于 Python 语言。其目前不支持 JSON。

9.5.9.4　Redis/Disque

由于 Redis 支持发布/订阅模型，因此 Redis 除了作为 NoSQL，还经常被用于消息队列来解耦应用服务器。Redis 之父 Salvatore Sanfilippo 新开源了一个分布式内存消息代理，其适用于 Redis 作为作业系统队列的场景，但采用了专用设计。其针对不可变的消息数据进行了优化。该设计独立、可扩展且具有容错功能。该设计兼具 Redis 的简洁和高性能，由 C 语言实现，是一个非阻塞型代理服务器。

Disque 的侦听端口为 7711，并重用了 Redis 的部分代码，且支持多主节点集群。Disque 的对应 Python 封装为 Pydisque。

9.6　业务与数据融合

通过 REST 接口和消息队列服务整合各种网络服务，这样对内可以构成微服务结构，对外可以与外界分享数据。这是一种网络设计的趋势，与此伴生的是安全问题，需要设计者特别关注。这方面也是大多数网站设计的重点之一。

9.6.1　网站权限管理

无论是何种类型的网站，都需要某种形式的访问权限管理。大致上有 RBAC 和 ACL 两种实现。

RBAC（Role Base Access Control）是面向角色的权限管理，即每个用户对应单一的角色，如注册用户、客服、管理员和超级管理员等，越复杂的系统角色就越多。针对不同用户角色，系统返回不同的视图（包括菜单、按钮等），并在访问不同 URL 路径（即网络资源）时做出限制。

ACL（Access Control List，访问控制列表）是面向资源的访问控制模型，如针对某一资源（如博客文章）的增删改权限。授权时，将特定“资源+操作”的组合记录在数据库中，并授权给特定用户。在复杂系统中，ACL 本身就是管理的难点之一。RBAC 的权限管理颗粒比较粗，但是实施比较简单。在实际操作中，也有混合型的权限管理模式。

一般而言，Web 框架并不提供权限管理模块，而是以扩展件的方式提供，如 PHP 中的 Zend_ACL、symfony-acl、ThinkPHP-RBAC、Yii-RBAC；Python 里的 Flashk-rbac、Django-rbac 等。原因很简单，权限管理和业务相关，确切地说其与业务和组织架构相关。Web 框架并不了解应用所需要的角色和资源管理策略，所以权限管理模块常常独立于 Web 框架。开发者首先需要掌握 RBAC/ACL 算法，并定义访问控制策略，然后在选定框架中实施，这有一定的工作量。

在 Python Web 框架中，需要首先了解 AOP 编程和装饰器的用法，才能够实现 RBAC/ACL 算法。

9.6.2 认证授权与计费

在企业级应用中，需要实现认证、授权和计费服务，即“Authentication，Authorization，Accounting”，缩写为 AAA。这是网络用户接入网络时的标配功能。远程用户拨号认证系统（RADIUS，Remote Authentication Dial In User Service）是这个行业的标配，并由 RFC2865/RFC2866 定义。开源免费的 FreeRADIUS 是目前的主流选择。此外，笔者发现新浪博客中有用 Python Twisted 实现的轻量级 RADIUS 设计。

有雄心成为物联网运营商的读者一定要了解这方面的发展。

9.6.2.1 RADIUS

RADIUS 是一种 C/S 结构协议。它的客户端最初就是 NAS（Net Access Server）服务器，任何运行 RADIUS 客户端软件的计算机都可以成为 RADIUS 的客户端。RADIUS 协议认证机制灵活，可以采用 PAP、CHAP 或者 UNIX 登录认证等多种方式。RADIUS 是一种可扩展协议，它进行的全部工作都是基于 Attribute-Length-Value 的向量进行的。RADIUS 也支持厂商自定义属性。

RADIUS 协议承载于 UDP 之上，官方指定端口号为认证授权端口 1812、计费端口 1813，以及标准化。

RADIUS 协议简单明确，且可扩充，因此得到了广泛应用，其适用于大型企业内部及 ISP（互联网服务供应商）采用的各种网络接入服务：包括普通电话拨号上网、ADSL 虚拟拨号上网、基于以太网和光纤的宽带虚拟拨号上网、VPDN（Virtual Private Dialup Networks，基于拨号用户的虚拟专用拨号网业务）、VPN 接入、IP 电话与移动电话预付费等业务。IEEE 提出了 802.1x 标准，这是一种基于端口的标准，用于对 Wi-Fi 无线网络的接入认证，在认证阶段时也采用 RADIUS 协议。所以，这种认证协议也可以用于物联网设备的入网认证。

9.6.2.2 NOC

NOC 是 Network Operation Center 的缩写，其是运营系统的一部分。NOC-Python 基于 Python Django 开发。其开发团队来自俄罗斯。这是一种面向互联网接入服务商（ISP）的可扩展、高性能和开源的运营支撑系统（OSS/BSS/BOSS）。NOC 支持多种网络设备，可以稍加修改并用作可营运物联网的业务支撑系统。

9.6.2.3 OpenLDAP

LDAP 服务器提供了访问、认证和授权的集中管理。它是企业级应用中常见的 AAA 组件。LDAP 基于 X.500 标准，但更简单，且可以根据需要定制。与 X.500 不同，LDAP 支持 TCP/IP，可以支持网络应用。

OpenLDAP 是轻量目录访问协议（Lightweight Directory Access Protocol，LDAP）的开源实现，在专门的 OpenLDAP 许可证下发行，并已经被包含在众多流行的 Linux 发行版中。

在 Python 中，有 python-ldap 扩展包作为 LDAP 的客户端。

9.6.3　OpenID

OpenID 是 Web 2.0 社交应用常见的第三方登录方式之一。OpenID 以用户为中心构建数字识别框架，具备开放、分散和自由的特性。它需要用户拥有自己的 URI（或网址）。其主要流程如下：

（1）用户希望访问 www.example.com；

（2）example.com 提示用户输入 OpenID URI；

（3）用户提供 OpenID URI，如 user.myopenid.com；

（4）example.com 跳转到 myopenid.com；

（5）用户在 myopenid.com 界面中输入用户名和密码；

（6）myopenid.com 向用户确认是否登录到 exmaple.com；

（7）用户同意后，跳转到 example.com；

（8）example.com 确认并认可用户鉴权，并访问网站中的资源。

9.6.4　OAUTH

另外一个更加重要的第三方登录方式是 OAUTH。其重要性体现在以下两点：

- OAUTH 在许多共享经济相关的物联网应用中起着引流和交易的重要作用；
- OAUTH 已经成为 REST API 的常见鉴权方式。

OAUTH 协议为用户资源的授权提供了安全、开放而建议的标准。OAUTH 授权不会让第三方触及用户账号信息，如用户名和密码。但 OAUTH 供应商授权提供的某些额外信息可能是不安全的，比如好友信息和内容信息等。其主要流程如下：

（1）用户在使用 example.com 时，需要从 mycontacts.com 导入联系人；

（2）example.com 把用户导向 mycontacts.com；

（3）用户在 mycontacts.com 登录账户，也可以采用 OpenID 方式鉴权；

（4）mycontacts.com 询问用户是否授权 example.com 访问其在 mycontacts.com 中的联系人信息；

（5）用户确定；

（6）mycontacts.com 把用户送回 example.com；

（7）example.com 从 mycontacts.com 获取用户联系人信息；

（8）example.com 将用户联系人资源成功导入。

各大互联网企业的 OpenID 和 OAUTH 或多或少都有些修改，需要特别注意其中的差异。

9.6.5 OpenID 与 OAUTH 的异同

OpenID 和 OAUTH 都是互联网社交化的成果。但这两者有何区别？总的来说，OpenID 是认证（Authentication），而 OAUTH 是授权（Authorization）。OpenID 是网站对用户进行认证，即用户向网站提供互联网 URL 以证明自己的身份。与 OpenID 相比，OAUTH 偏重授权，只是授权的前提是认证。其流程如下：A 网站给 B 网站一个 Token，B 网站凭着该 Token 可以访问某用户在 A 网站中的资源。

OpenID/OAUTH 之所以容易混淆，主要是由于第三方登录本质上仅仅是认证（Authentication），两者都需要经过认证环节：

- OpenID 认证，通过交换属性方法获取用户名和其他属性；
- OAUTH 认证，获取到 Token 后，用 Token 获取用户名和其他信息。

国内互联网企业往往都已经平台化，所以 OAUTH 的使用更普及。笔者选择 Cyclone 作为自己的 Web 框架，原因之一在于它内置了 OpenID/OAUTH 的支持，并适配了 Google/Facebook/FriendFeed 的 API。如果增加国内企业的第三方登录，修改工作量不大。

利用 OpenID/OAUTH 的第三方账户登录网站后，许多网站依然要求提供更多个人信息并注册新账号，这是网站自己的实施策略。

如果物联网企业自己提供单点登录，那么该企业的网站扮演的是 OpenID/OAUTH provider（供应商）角色，还需要在服务器端增设 OpenID/OAUTH provider 模块。可以基于 python-openid 和 python-oauth2 模块，定制 OpenID/Oauth 服务器，并提供客户端设计给第三方开发者使用。

9.6.6 社交化硬件

利用强大的社交生态构建自己的第三方登录系统，可以实现产品的社交化和快速推广。

社交网络刚出现时就已经实现了与物联网的融合。Twitter 的开放 API 曾被用作远程开关的通道就是一个典型例子。而国内的腾讯则将微信和硬件进行了整合。

社交账户不仅限于 WeChat、Taobao、Weibo，还有 Lineup、Google+、Hotmail、Skype、Yahoo、Twitter 和 Facebook。笔者认为，社交硬件应该在共享资源方面下功夫，而非在朋友圈中“晒”各种锻炼数据等。最近火热的共享单车、公寓、民宿应用中大量采用微信和支付宝的 OAUTH 方式登录，这是社交化硬件的新发展趋势。

社交账户的第三方多采用 OpenID/OAUTH 认证，但是某些公司的认证方式会有一些自己的修改以及其他的方案，读者需要参考其各自的 API 文档说明。

以上分别介绍了针对公网、企业内网、运营商和互联网的 AAA 组件，读者可以根据各自的需求选用，并集成在物联网应用系统中。

9.7　Web 开发框架

一个完整的物联网应用实际上需要由多台服务器组成：

- 接入服务器由 MQTT 或 socket server 提供；
- 接入服务器的配置管理服务器由 Web Server 提供；
- 与 APP/第三方服务器的数据分享依赖于 Web 服务器提供 REST API 以及 SSE 推送；
- 与业务、组织机构管理、OpenID/OAUTH 相关的业务应用服务由 Web Server 提供；
- 其他辅助服务器，如负载均衡、数据库、文件服务器、消息队列等。

可以发现，在物联网服务器组成中，依然离不开 HTTP Web Server。Web 开发实际上是 IoT 开发的一部分。所以，我们有必要重温一下 Web 服务器的设计。

9.7.1　MVC 模型

网站开发有自己的设计模式：MVC（Model-View-Controller）。

- Model，模型建模：其指将对象资源间的关系映射到数据库表格中。
- View，视图：其指我们看到的网页外观，同时还需要使用网页模板。
- Controller，控制器：每个网页需要实现的功能，由对应于该网页路径的函数实现，该函数就是控制器，用于整合模型（数据库）和视图（网页）。

由于前端网页的迭代更新，因此最新的 Web 开发趋势是 MVP、MVVM，这可以被理解为 MVC 的衍生模型。

9.7.2　Web 开发流程

Web 开发已经非常成熟，其设计流程可以按照以下步骤进行：

（1）了解客户的需求，进行数据建模（Model），确立数据库对象之间的关系；

（2）了解客户对于页面导航流程的需求，建立导航关系图，这和视图（View）有关；

（3）根据前者，规划定义 URL 路由（Controller）。

规划 MVC 时需要注意：

（1）根据模型的逻辑关系来设计 URL 路由，尽量扁平化，减少逻辑层次关系，不要被导航关系所迷惑。

（2）使用模型的对象（名词）和动作（动词）来规划路径，减少的逻辑关系采用 URL 参数

来补充。

（3）视图设计包括 HTML、CSS、JavaScript（用户互动和动画）和 AJAX 调用。

（4）视图设计本身需要模板化，将其分割为若干可复用部分：页眉、脚注、内容、导航等，这些都可以通过模板复用。

（5）将手头的 UI 组件做成快速原型提供给客户做选择，减少客户自定义修改的机会。

在视图设计中需要增加以下步骤：

（1）在定义 URL 路径时，需一并规划 AJAX 所需的 REST API 路径；

（2）与客户讨论 UI 时，需要加入动画和动作交互设计概念；

（3）在视图设计阶段，规划模板时需要明确加入对于 HTML class/id 的命名规则；

（4）JavaScript/AJAX 动作可以减少页面间的流转，减少带宽消耗，设计师可以酌情考虑。

顺便提一下 HTML 静态网页。许多情况下，文档之类的内容仍需要采用静态网页技术。现在笔者不再“设计”静态网页，而是按照 reST 格式文档编写网页内容，然后用 Sphinx 编译成 HTML 网页，最后将编译后的 HTML 网页映射到动态 Web 站点的 URL 中。至于风格和皮肤，可以通过 CSS 进行定制。这适合一些文档性质的 Web 内容。

9.7.3 Python Web 百花齐放

探讨选择何种 Web 开发框架的话题很容易在开发者之间沦为“口水战”。先不论编程语言间的选择，光 Python Web 开发框架之多，已经让人无法进行完整评估。笔者将这些 Python 框架按照同步和异步模型进行归类。常见的 Python Web 框架请查看表 9-9。

同步方式就是在用户请求内容的过程中，如果涉及数据库等慢速任务，例如“慢 SQL”查询，都一直等待，直至数据返回。所以，一个较长的用户请求会阻塞后续请求。异步方式是采用内部调度器加回调的方式去实现异步通信。这样，一个任务可以不用等待，而可以直接服务于下一个请求，等之前的任务完成后再分别返回给客户端。其系统整体反应敏捷。在各大语言中，都出现了异步模式的框架。与其他编程语言相比，Python 采用异步模式较早。

表 9-9 Python Web 框架

名　称	模　式	简　介
Zope	同步	Python 版本的 J2EE，历史悠久，支持多种安全框架，适合企业开发
Django	同步	Python 的快速 Web 构建系统，一站式 Web 框架
Flask	同步	轻量级 Web 框架，主要由 Werkzeug 和 Jinja2 构成
CherryPi	同步	开发简单，需要自行选择其他组件，适合为其他应用提供 Web 管理界面
Web.py	同步	一套非常简单的框架，适合做其他系统的 Web 管理界面
Tornado	异步	极轻量级、高可伸缩性和非阻塞 I/O 的 Web 服务器软件

续表

名　称	模　式	简　介
Twisted	异步	Python 通用网络编程框架
Klein	异步	基于 Twisted 的轻量级 Web 服务器
Cyclone	异步	基于 Twisted 的 Web 服务器，较为完整

虽然异步框架的性能优秀，但 Twisted 之类的异步设计中也还有些局限。同时，同步框架中存在大量的现成组件设计可以使用。利用 gevent 可以很好地改造升级同步框架。所以笔者在最近的一些设计中，往往采用同步加异步的方式来组合设计。一方面利用 Django/Flask 的扩展件和现成设计，另一方面采用异步框架来实现异步通信。两者间采用数据库、消息队列和 Web API 方式进行耦合。这种方式很适合物联网将设备云和其他应用（CRM/ERP/BOSS 等）分离的架构。具体实现代码可以参考 *Lightweight Django* 一书。

9.7.4　Zope

Zope 是一个开源代码的 Web 应用服务器，目前有 Zope2/Zope3 系列版本。Zope2 特别适合脚本开发人员，可以通过浏览器快速构建一个应用；Zope3 采用了现代设计模式，其是一个基于组件架构的应用服务器。业界称之为 Python 版本的轻量级 J2EE 框架。

Zope 开发历史悠久，但是其遗留了许多有用的组件。比如 Zope.interface 就在许多 Python 框架中使用。Zope 允许并鼓励第三方开发者打包和分发应用程序。因此，已经积累了大量的现成产品组件。大多数组件都是自由且开源的工程。Zope 创建的应用程序可以直接通过 Zope 企业对象（ZEO）进行扩展。其好处是可以在多台计算机中部署 Zope 应用程序，而不需要修改代码。Zope 允许开发者使用浏览器中的 Zope 管理界面（ZMI）来创建 Web 应用程序。使用统一的 Web 界面，可以让开发者安全地并行开发。Zope 提供了多种和可扩展的安全框架，可以轻松结合多种权限认证系统。通过内置模块，其可以同时支持 LDAP、Windows NT 和 RADIUS。

9.7.5　Django

Django 是 Python Web 中另外一个重要的框架。它最初是加拿大劳伦斯出版集团用于管理旗下新闻网站内容的设计。本质上，Django 更适合作为内容管理系统（CMS）软件。该设计于 2005 年 7 月在 BSD 许可证下发布。在 PyPI 官网索引中，可以找到一大堆与 Django 相关联的包。

Django 的 MVC 设计十分优美。

- 对象关系映射：ORM 将模型与关系数据库连接起来，得到一个非常容易使用的数据库 API，同时也可以在 Django 中使用原始的 SQL 语句。
- URL 分发：使用正则表达式匹配 URL，可以设计任意的 URL，没有框架的特殊限定。

- 模板系统：使用 Django 强大而可扩展的模板语言，可以分离设计、内容和 Python 代码，并且具有可继承性。
- 表单处理：可以方便地生成各种表单模型，实现表单的有效性检验。可以方便地从模型实例生成相应的表单。
- 缓存系统：可以挂载内存缓冲或其他的框架，实现超级缓冲以实现所需要的粒度。
- 会话：用户登录与权限检查，快速开发用户会话功能。
- 国际化：内置国际化系统，方便开发出多种语言的网站。
- 自动化管理界面：不需要创建人员管理和更新内容。Django 自带一个管理界面。

Django 是一个大而全的 Web 框架，但是定制起来需要一定的学习时间。

9.7.6 Flask

Flask 是一个使用 Python 编写的轻量级 Web 框架。其 WSGI 采用 Werkzeug，模板引擎采用 Jinja2。它被称为微型架构，因为其使用非常简单的核心以及功能丰富的扩展。虽然 Flask 内核没有默认的数据库、验证工具等，但是却通过扩展加入了 ORM、验证工具、文件上传、开放式身份验证，可以实现非常强大的 Web 应用。Flask 将这些扩展命名为 blueprint。

Flask 也兼容 Google AppEngine 等 PaaS 平台。因为 Flask 是一个同步堵塞式的 Web 框架，必要时可以配合 gevent+gunicorn 来增强性能。Flask 支持 Web 应用的热更新，一旦用户更新了 Web 应用程序，Flask 会自动加载，无须用户手动重启。

simple_flask.py：

```
#!/usr/bin/env python

from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

在代码中，Flask 采用了装饰器@app.route("/")定义了 URL 路径。

```
$pip install Flask

$python simple_flask.py
```

如需查看网页内容，可在浏览器中输入：

```
http://localhost:5000/
```

9.7.7　gevent 提升性能

Python 的 GIL 使得多线程无法充分利用多核服务器，这一直是 Python 开发的瓶颈。围绕着这个问题出现了各种各样的解决方案。Web 站点一旦形成规模，就可使用 gevent+ Flask/Django 来提高系统性能。

flask_gevent.py：

```
from gevent.wsgi import WSGIServer
from pure_flask import app
http_server = WSGIServer(('', 8888), app)
http_server.serve_forever()
```

gevent 底层实现却是异步 I/O，而且对于应用是透明的。通常异步框架性能要高过同步阻塞型框架，通过 gevent 来改进同步框架，无须修改应用代码即可以提升 Web 性能。此外，gevent 还支持使用多进程模式，可充分利用多核的并行处理能力。gunicorn 自带 gevent，可以替代 Web 框架中的 WSGI 服务器。

9.7.8　异步 Web 框架 Tornado

Tornado 是 FriendFeed 开发的 Python Web 框架。FriendFeed 被 Facebook 收购后，Tornado 被提交给开源社区使用。Tornado 是非阻塞式服务器，速度相当快。作为一个理想的实时 Web 框架，Tornado 的每个 HTTP 连接都是长连接。所以其很适合新型 Web 的开发，如支持实时消息推送服务，或者基于 HTTP 的物联网应用。

Tornado 经常被拿来与 Twisted 进行对比。Tornado 区别于 Twisted 最重要的一点是，Tornado 是完整的 Web 框架，而 Twisted 在所支持的互联网协议种类上要远多于 Tornado。Twisted 虽然支持 HTTP 协议，但却不是一个完整的 Web 框架。

9.7.9　异步网络框架 Twisted

前面反复提到了 Twisted，这个框架最初是为游戏而开发的平台。其后来逐渐拓展到不同的应用。虽然 Twisted 的开发历史很长，但是一旦有新的网络协议，利用 Twisted 就可以很容易地实施。Twisted 属于 Python 的“常青树”之一，也是笔者用于为各类客户项目工程提供定制协议的基础。

然而，Twisted 依然不够完美：

- Twisted 没有提供完整的 Web 框架，需要更加抽象的 Web API；
- Twisted 的异步编程入门需要先学习 yield 语句、生成器、I/O 复用、装饰器和 AOP 编

程概念；

- 使用 CPython 时性能受限，需要 PyPy 加速。

笔者总的设计思路是，基于 PyPy+libuv/pyuv 加速 Twisted 应用，构建多台多核平行的工作服务器，前端再使用负载均衡。这样可以构成比较完整的接入服务器设计。

9.7.10 异步 Web 框架 Cyclone

Cyclone 是基于 Twisted 的 Web 框架，几乎和 Tornado 一样丰富。我们可以简单地理解为 Cyclone = Twisted + Tornado API。它提供了更加完整的 Web 相关功能。Cyclone 的特性如下：

- 异步 Web 框架；
- 支持 Tornado/Bottle API 风格编程；
- 数据库支持 SQLite/Redis/MySQL/PostgreSQL；
- 内置电子邮件用于注册、重置密码和报警等；
- 支持国际化（internationalization）和本地化（localization）；
- 支持 HTTPS 和 TLS/SSL；
- 支持主流 OpenID/OAUTH 认证，包括 Google/Twitter/Facebook/FriendFeed；
- 丰富的例子，并带有启动工程。

许多 Python Web 框架的 URL 路由定义分散在各个 URL Handler 函数的装饰器中，查找比较麻烦。而 Cyclone 则将 URL 路由定义集中在路由列表中，这种集中管理方式容易寻找，可以避免重复定义 URL。

9.7.11 静态网页

Python 自带服务器就可以完成静态网页的搭建。几乎所有的 Web 框架都支持简单的静态网页，包括 Twisted/Cyclone。

```
twistd web --port 80 --path ./public/html
```

以上语句就是利用 twistd web 子命令，以“/public/html”作为根目录，首页为 index.html，构建一个最简单的 80 端口 HTTP 服务器。这很适用于文档、多媒体资源或者单页 JavaScript 网页的演示型、媒体型和内容型站点。

9.7.12 TLS 安全网页

互联网从诞生之初就面临各类安全挑战，许多主流网站都采用 HTTPS 来保护 HTTP 服务，HTTP2 标准也将 HTTPS 作为事实标准提供。TLS 成为网站标配。Ngnix/Apache/WSGI 或其他

Web 框架有着不同的 TLS 证书部署方式和流程。在高并发架构中，往往使用 Ngnix 构建前置负载均衡代理服务器，将 Python 作为后端工作服务器。那么只需将 TLS 证书推送到 Ngnix 服务器中，后端工作服务器无须任何配置就可以实现高负荷的服务器集群。但是小规模应用依然需要在 Python Web 服务器端部署 TLS 证书。

下面的例子以 Cyclone 为例介绍 TLS 证书部署。在 Cyclone.io/GitHub 官网中包含三个文件：

```
helloworld_simple.py
helloworld_ssl.py
mkcert.sh
```

构建流程如下：

（1）产生自颁证书，或者采购 TLS 证书；

（2）运行 helloworld_ssl.py；

（3）浏览 http://localhost:8443。

9.7.12.1　构建 TLS 证书

首先利用 mkcert.sh 脚本创建自颁证书和密钥，并放置到合适的路径中。

mkcert.sh：

```
#!/bin/bash

echo -- key
openssl genrsa -des3 -out server.key 1024

echo -- csr
openssl req -new -key server.key -out server.csr

echo -- remove passphrase
cp server.key orig.server.key
openssl rsa -in orig.server.key -out server.key

echo -- generate crt
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

注意　由于这是自颁证书，因此浏览器会提示证书非法。我们可以选择忽略，将该地址加入安全例外中即可。自颁证书不推荐使用于实际运营的生产环境中。同时开发者需要注意，在开发手机 APP 中，自颁证书和第三方证书可能在开发方式、调用的类等方面有所区别。

9.7.12.2　源码与运行方式

首先，我们来看看最简单的服务器源码。

helloworld_simple.py：

```
#!/usr/bin/env python
# coding: utf-8
#
# Run: twistd -n cyclone --ssl-app helloworld_simple.Application
# For more info: twistd -n cyclone --help

import cyclone.web

class MainHandler(cyclone.web.RequestHandler):
    def get(self):
        self.write("Hello, %s" % self.request.protocol)

Application = lambda: cyclone.web.Application([(r"/", MainHandler)])
```

在最简单的 Web 服务器代码中，对于 TLS 证书没有任何设置。可以采用插件方式运行，在终端中输入：

```
twistd -n cyclone --ssl-app=helloworld_simple.Application
```

作者 Alexandre Fiori 还提供了另外一个服务器和相对应的运行实例。

helloworld_ssl.py：

```
#!/usr/bin/env python
# coding: utf-8

import cyclone.web
import sys
from twisted.internet import reactor
from twisted.internet import ssl
from twisted.python import log

class MainHandler(cyclone.web.RequestHandler):
    def get(self):
        self.write("Hello, %s" % self.request.protocol)

def main():
    log.startLogging(sys.stdout)
    application = cyclone.web.Application([
        (r"/", MainHandler)
    ])

    interface = "127.0.0.1"
    reactor.listenTCP(8888, application, interface=interface)
    reactor.listenSSL(8443, application,\
                      ssl.DefaultOpenSSLContextFactory("server.key",\
                                                       "server.crt"),\
                      interface=interface)
    reactor.run()
```

```
if __name__ == "__main__":
    main()
```

在相对复杂一点儿的 Web 服务器中，配置了明文端口和 TLS 端口。由于官方例子提供了配置脚本，因此可以直接执行脚本运行服务器。

```
python helloworld_ssl.py
```

注意　一般来说，Web 服务器明文 HTTP 端口为 80，而 TLS 保护的 HTTPS 端口为 443。在设计环境中，可以分别使用 8080 或 8443 来替代。在生产环境中，推荐关闭 80 HTTP 端口。

9.7.12.3　bash 脚本

对于以上两种方法，读者可以自行选用。因为 Twistd TAC/Twistd Plugin/Cyclone/systemd 等多种运行方式已经够多了，所以笔者不太愿意由于 TLS 证书问题而将源码设计复杂化。

参考文档，笔者编写了两个启动脚本，无须修改 Web 源码，可分别启动无加密的 HTTP Web 和安全的 HTTPS Web。

plain_daemon_runit.sh：

```
#!/bin/bash

export PYTHONPATH=`dirname $0`
twistd -yn cyclone -p 8888 -l 0.0.0.0 \
       -r easyiot.web.Application -c easyiot.conf $*
```

ssl_daemon_runit.sh：

```
#!/bin/bash
export PYTHONPATH=`dirname $0`
twistd -yn cyclone --ssl-app=easyiot.web.Application -c easyiot.conf $*
```

9.8　物联网安全

网站安全的重要性毋庸置疑。虽然我们无法预知未来还会出现何种攻击手段，但是了解现有的攻击方式，避免一些低级错误是可以做到的。除了通过 SSL/TLS 增加端对端的安全性，系统上线前还可以通过众包形式请安全专员做安全渗透测试。

就目前来说，我们可以从多个层面加强系统安全性。这里有个基本的原则：不要相信来自用户的任何请求。

9.8.1 物联网安全现状堪忧

物联网的环节比传统互联网更长，其也更容易受到攻击。最糟糕的是由于边缘节点的计算资源和通信资源有限，因此其简直可以用**漏洞百出**来形容。目前，物联网最常见的攻击方式如下。

- 伪装节点上传数据。比如上传极端温度指数，触发自动灭火装置喷淋，造成实际损失；另外，冷链管理失效对于乳制品和疫苗企业造成的损失也是巨大的。
- 耗尽设备的电池储备。通过恶意攻击不断消耗 WSN 设备的有限电池储备，造成节点和网络瘫痪。
- 远程下载黑客固件。通过下载修改过的固件，造成物联网设备或数据的损失。
- 盗取用户个人资料。通过远程摄像头拍摄私人照片，获取联系人信息，通过伪基站发送短信进行“社工”攻击。
- 远程劫持用户设备。比如远程劫持手机加密存储器勒索赎金，远程攻击联网汽车获得控制权，以及干扰汽车电子遥控门禁 RKE/PKE 系统等。

所以，不要以为只读设备就安全；不要以为远程固件更新就安全；不要以为私有协议是安全的；不要以为自己比黑客更加聪明，多少技术类相关的网站被攻破就是一个例子，就连美国 NSA 都被黑客攻破过。

互联网、移动互联网、物联网的安全是一项长期的攻防战，千万要紧跟时代。

9.8.2 操作系统安全

PaaS 在系统安全性上超越了 IaaS，因为普通开发者根本没有访问服务器操作系统的权限。由于大多数 IaaS 均基于 Linux 系统，因此，关于服务器操作系统安全方面的问题和解决方案，例如如何正确使用 Linux、如何合理配置权限、如何配置密码和定期修改密码等安全策略，读者可以参考 Linux 管理方面的书籍。网络上也有 Linux 安全加固方面的文章可供读者借鉴。其主要的使用原则如下：

- 锁定多余账户；
- 设置账户口令策略；
- 禁止 root 之外的超级用户；
- 限制能够提权为 root 的用户能够做的事情；
- 检查 shadow 中的空口令账户；
- 关闭无关的系统服务；
- 设置合理的文件权限；
- 使用 SSH 做远程管理；

- 限制能够登录 SSH 的 IP 地址；
- 禁止 root 用户远程登录；
- 限定信任主机；
- 屏蔽登录 banner 信息；
- 设置登录失败次数；
- 限制 FTP 登录用户；
- 设置 Bash 命令保存历史；
- 设置日志策略。

做到以上这些可能还不足以应对黑客的攻击，尤其是暴力攻击。我们需要定期查阅日志中暴力攻击的记录，以掌握黑客攻击的趋势，并有效加固安全策略。

9.8.3　数据缓存与数据持久层安全

SQL 注入是最基本的持久层侵入手段。所谓 SQL 注入，就是通过 SQL 命令将非法字符插入 Web 表单提交内容中，或是域名和页面请求的查询字符串中，最终达到欺骗服务器来执行恶意 SQL 命令的目的。为此，黑客攻防里有一个专门的工具：sqlmap。该工具自称为全自动 SQL 注入及数据库提权工具，采用 Python 编写，全部开源。没错，Python 不仅是开发者的最爱，也是黑客的最爱。

sqlmap 是一把双刃剑，它可以用来测试自己网站的缺陷以防止 SQL 注入，也可以用来攻击其他网站。但是相当多的网站认为 SQL 注入不会发生在自己的身上，所以利用 sqlmap 来测试一把是很有必要的。

除了 SQL 数据库需要注意安全外，Redis 作为风头强劲的键值型 NoSQL，默认情况下无须密码即可以访问数据库，所以我们需要额外注意增设密码。由于 Redis 性能强劲，因此也很容易使用穷举法破解。所以当 Redis 数据库作为单独的服务器使用时，其对外的密码一定要非常长而复杂。

以前爆出的大批 MongoDB 被人大量破解并勒索赎金的事件，究其原因就是由于黑客利用了配置有误且可公开访问的漏洞，无须具备相应的管理员凭据即可以展开攻击。网站开发者、管理员需要密切关注此类安全事件，部署对应的安全措施。

9.8.4　Web 框架与容器安全

Web 容器的概念来自 Java，Python 中的大多数是 Web 框架。虽然我们出于开发的便利，使用了大量的 Web 框架，但是这些框架或容器也都有各自的缺陷，需要不断地跟进和部署安全补丁。挑选大家都使用的主流 Web 框架，安全性是一个很重要的考虑点。我们可以在网络上寻找

服务器系统及软件常见漏洞列表来看看自家服务器及服务器容器的安全风险。

笔者自己的服务器上线未做任何推广。但上线没有多久，笔者就发现不断有人尝试利用 Apache 的 TRACE/CONNECT 命令来攻击服务器。所幸，笔者没有采用 Apache+PHP 的设计，而是采用了 Python Twisted。在实际设计中，如果采用了 Web 框架，设计者能够做的事情并不多。在设备接入的套接字服务器中，笔者增加了有针对性的设计。过滤到此类攻击时，应用程序可采用直接挂断连接或以超时连接挂断方式应对，以避免攻击者通过连接恶意占用连接资源。

9.8.5 远程加载风险

ImageMagick 是命令行下的图形处理程序，它被许多网站用于处理用户上传的图片，如缩略图、剪裁和转换等。但是，ImageMagick 爆出了可以实现远程加载的漏洞。即，黑客可以上传一些“特制”的图片，并可以实现操作系统提权，而且可通过网络将 shell 暴露给远程黑客。接下来的事情可想而知：系统配置泄露，用户数据库泄露……

与此类似，许多软件如 Adobe Flash、微软、Oracle 的某些软件，都存在远程加载漏洞。

凡是网站对外的用户接口，包括 HTML Form 以及 HTTP URL、文件上传、图片上传都要做严格的审查，并且注意对应软件的安全升级。

9.8.6 Web 前端安全

如果说服务器的其他端口都封闭、安全策略做到位的话，那么 Web 前端是必须暴露在外界的端口，我们必须额外注意。

9.8.6.1 XSS

XSS（Cross-Site Scripting），即跨站脚本攻击，它是注入攻击的一种。其特点是不对服务器端造成任何伤害，而是通过一些正常的站内交互途径，例如发布评论，提交含有 JavaScript 的内容文本将攻击代码提交到 Web 服务器中。当其他用户访问到这些页面时，会遭到页面内恶意脚本的攻击。这时服务器端如果没有过滤或转义掉这些脚本，而作为内容发布到了页面上，其他用户访问这个页面的时候就会运行这些脚本，并对网站用户造成损害。

9.8.6.2 CSRF

CSRF（Cross-Site Request Forgery），即跨站伪造请求。顾名思义，CSRF 是伪造用户请求，即冒充用户在站内的正常操作。

绝大多数网站是通过 cookie 等方式辨识用户身份的，包括使用服务器端 Session 来辨识用户的网站。因为 Session ID 也是大多保存在 cookie 里面并以此为凭证予以授权的，所以要伪造用户的正常操作，最好的方法是通过 XSS 或超链接欺骗等途径，让用户在本机，即拥有 cookie

的浏览器端发起用户无法察觉的请求。

从严格意义上来说，CSRF 不能被归入注入攻击，因为 CSRF 的实现途径远远不止 XSS 注入这一条。通过 XSS 来实现 CSRF 易如反掌，但对于设计不佳的网站，一条正常的 URL 超链接都能造成 CSRF。

9.8.7　传输层安全

所谓传输层安全（Transport Layer Security）就是在 TCP/UDP 传输层实现安全传输通道，可以避免中间人攻击。在安全通道之上，甚至可以使用明文传输。此类技术最早就是 SSL（Secure Socket Layer）。SSL 协议位于 TCP/IP 协议与各种应用层协议之间，为数据通信提供安全支持。SSL 协议可分为以下两层。

- SSL 记录协议（SSL Record Protocol）：它建立在可靠的传输协议（如 TCP）之上，为高层协议提供数据封装、压缩、加密等基本功能的支持。
- SSL 握手协议（SSL Handshake Protocol）：它建立在 SSL 记录协议之上，用于在实际的数据传输开始前，通信双方进行身份认证、协商加密算法、交换加密密钥等。

TLS（Transport Layer Security）是 SSL（Secure Sockets Layer）的升级替代品，这是为网络通信提供安全及数据完整性的一种安全协议。

在现在的主流设计中，先用 SSL/TLS 进行彼此认证，交换密钥。然后在建立的安全通道基础上，利用交换后的密钥采用 AES 等对称加密算法进行加密传输。在这种指导思想下，产生了 SSH、SFTP、HTTPS、IMAP 等一系列服务。物联网协议中的 MQTT 也支持 TLS 传输，CoAP 也可以支持 D-TLS 等衍生版本。标准制定者对于安全是非常重视的，这值得开发者仔细研究。

9.8.7.1　双向认证的必要性

不仅仅 SSL/TLS 要实现双向认证，许多物联网应用领域如 Mifare RFID 产品线也基于对称算法实现了双向认证。但互联网的安全要求更高，使用了非对称算法的双向认证，以及对称算法的内容加密。

为什么我们需要验证双方的身份，而不仅仅验证客户端身份

这是因为中间人攻击在一次通信会话中既可能伪装成设备端/客户端欺骗服务器信任，也可能伪装成服务器欺骗设备端/客户端信任。新闻报道中的 GSM 伪基站，就是因为 GSM 系统只实现了基站对于终端的单向认证，而终端却无法识别基站的合法性，造成了 GSM 用户的大量实际损失。

9.8.7.2　Web API 更需 TLS 保护

如果没有 TLS 保护，Web API 设计需要在通道不安全的基础上保证服务器安全、客户端安

全和数据传输安全，且还要保证 API 的无状态特性，设计复杂度非常高。

在 Web API 中，每次交易时我们都必须做一次彼此认证的流程，并不得使用“明文”传输任何敏感信息。如果采用 cookie 之类的设计，会破坏 HTTP 的无状态特性，也不见得有多安全。

在传统的用户登录流程安全设计中，采用 username + password。在 Web API 上对应的是 API-key + API-secret。笔者对网络安全中滥用“key/secret”十分无奈。由于面向人的场景中，username/password 非常通俗易懂，所以下面会交叉比较 username/password 和 API-key/API-secret。

在服务器设计中，username 和 password 最原始的方式是以明文保存在服务器数据库中。因为大多数人只能够记住少量的 username/password 组合，而且不会经常改变，这也是网站数据库遭到 SQL 注入并撞库后，许多人的 username/password 组合可以用来试探并攻击该用户名下其他网络服务的原因。

为了避免明文存储 password，在许多开发框架中开始引入 password hash。客户登录时，提交 username/password，但 password 在网页端采用哈希算法计算出 password hash。服务器将用户提交的 username/password hash 与数据库进行比较，结果一致就可以登录。除了 password hash 之外，甚至还要“加盐”。所谓加盐不是炒菜，而是为避免某种逆向分析算法而加设的随机数。再加上随机验证码和错误限制，通过随机数信息的引入，这样的方式至少确保了 username/password 组合不会那么容易地被获取。

Web API 与用户登录流程有类似的地方，但是依然存在差异。首先，Web API 的使用频率远高于用户登录。如果攻击者记录了所有的 password hash，那么就有可能在大量数据基础上破解 API-secret。其给服务器端设计者带来的问题在于，需要服务器在不知道 API-secret 明文的基础上判断对方的 API-secret 是否正确。但如果 API-secret hash 一直保持不变，那么和明文也没有任何区别。所以，API 哈希算法必须基于 API-key、API-secret、timestamp、random 四个参数进行计算和比较。

这方面的设计原则可以参考 *RESTful Web Services* 一书，作者为 Leonard Richardson、Sam Ruby。此外，参考阿里云的 API 网关和 AWS 的 API 设计，读者也可以构建自己的 API 权限管理。

综上所述，在非安全通道上保证服务器和客户端的安全流程，其实现的代价很高，占用了大量的计算资源。此外，如何下发和配置 API-key 和 API-secret 也需要设计一个安全认证流程。这还不如在传输实际数据之前完整地走一遍 SSL/TLS 双向认证流程，然后开启会话，在彼此信任的基础上，在安全加密通道中，以明文方式传输 API-key。这样大大减轻了应用层的复杂度。另外一个很重要的考虑是，SSL/TLS 是安全标准，比我们自己攒出来的“补丁”设计接受度要高，而且经受了大量网络验证。

综上所述，SSL/TLS 提升了数据传输通道安全、简化了系统设计、有多种标准化方案可以实施，是一个非常值得的投资。不过，TLS CA 证书较贵。好在现在许多云计算和第三方颁证机构可以免费提供 CA 证书，开发初期我们可以使用这些免费证书，业务规模发展后，再购买

全功能的 CA 证书。TLS 的另外一个顾虑就是深嵌入式系统的资源受限问题。这些设计实施 TLS 中的非对称算法非常困难。可以选择硬件加速或者稍简单的非对称算法来实现认证，然后使用标准 AES 实现内容加密。虽然不是标准 TLS，但是多少参考了 TLS 的安全模型。

9.9　服务器交付

任何设计都有交付的阶段。如何轻松地交付服务器设计，甚至实现持续集成也是一个需要考虑的环节。

9.9.1　虚拟机交付

服务器的配置管理是一件复杂的工程管理项。为了保持开发和生产环境尽可能一致，我们的服务器交付往往依赖虚拟机或者容器技术。所谓虚拟机就是以 VirtualBox 或者 VMware 的文件格式进行交付。这样，我们只需要配置一次，就可以运行在其他服务器中。一些 Linux/Web 开发教程现在就直接交付 VMware Appliance 让大家使用。不同虚拟机，如 VirutalBox 和 VMware 之间采用 OVF（Open Virtualization Format）和 OVA（Open Virtualization Application）格式进行交付。

但是不同云服务供应商对于虚拟机的支持是不同的，至少国内的许多供应商还不支持这种方式。网站开发者还需要自己手动或者编写脚本来配置环境。经过了解，笔者得知阿里云的服务器快照就可以作为自定义映像复制到多个服务器实例中去。配置完一台服务器，就可以快速复制到多个服务器实例中，然后利用运维工具根据不同服务器进行个性化配置。

在服务器相关的设计中，虚拟机往往是把服务器操作系统如 Ubuntu/Centos 一起打包，其映像文件往往都在 GB 数量级。而容器技术作为一种新的轻量级虚拟技术，交付文件大小在几百个 MB 左右。容器技术得到了许多互联网公司和开发者的推崇。

9.9.2　Docker 容器交付

Docker 容器使用 Google 的 Go 语言开发。该容器是互联网开发和运维团队的热门话题之一。Docker 利用 Linux LXC 实现的轻量级虚拟化，几乎没有任何额外开销。另外，Docker 容器的启动与停止在几秒内完成。Docker 的使用场景如下。

- 简化配置：Docker 可以将运行环境和配置放在代码中部署，同一个配置可以在不同环境中使用。
- 代码管理：由于开发、测试和生产环境可以共享一个配置，因此简化了代码的管理。
- 提高开发效率，避免了在不同代码中的切换。

- 隔离应用：可以将单体服务器应用分拆为单个微服务。
- 整合服务区：多个应用可以共享没有使用的 RAM。
- 调试能力：Docker 提供了一些调试能力。
- 多租户环境：由于天然隔离应用，实现多个实例的隔离就是多租户的应用。
- 快速部署：Docker 部署是秒级，虚拟机为分钟级，物理机就更加慢了。

Docker 的核心收益就是应用隔离、轻量虚拟化，这简化了配置与运维难度和工作量，可以实现持续交付和多租户平台。

目前国内多家云计算供应商提供 Docker 容器服务。其中部分供应商提供免费版本，可以用于评估：

- 阿里云 Docker 容器的公测阶段是免费的。
- 希云（cSphere）提供三个版本的容器服务——免费、免费使用和收费三种版本。
- DaoCloud 提供三个版本的容器服务：社区版，专业版，企业版。其中，社区版免费。

Docker 与微服务架构

微服务指的是开发单个小型但有业务功能的服务，每个服务都有自己的处理和轻量通信机制，可以部署在单个或多个服务器上。同时，微服务也指的是松耦合、有一定上下文的 SOA（面向服务架构）。Docker 实际上加速了微服务架构的流行和普及。

1）微服务架构的优点

- 每个微服务都很小，能聚焦指定的业务功能或业务需求。
- 微服务能够被小团队单独开发。
- 微服务是松耦合、有功能意义的服务，无论开发阶段还是部署阶段都是独立的。
- 微服务能使用不同的语言开发。
- 微服务允许以灵活方式集成自动部署，并充分利用持续集成工具。
- 团队的新成员能够更快投入生产。
- 微服务易于被开发人员理解、修改和维护，小团队能够更关注核心优势。
- 微服务允许利用并融合最新技术。
- 微服务是业务逻辑的代码，不和 HTML、CSS 或其他界面组件混合。
- 微服务能够即时扩展。
- 微服务能部署在中低端配置的服务器上。
- 易于第三方集成。
- 微服务都有自己的存储能力，可以有本地数据库，也可以有统一数据库。

2）微服务架构的缺点

- 微服务架构可能带来过多的操作。

- 需要 DevOps 技巧。
- 可能需要更多工作量。
- 分布式系统可能比较复杂、难以管理。
- 分布部署，跟踪问题难。
- 当服务数量增加时，管理复杂性也随之增加。

3）微服务架构适合哪种情况

多数应用，如 Web 应用、移动互联网、智能电视、可穿戴设备和智能设备都适用于微服务架构。

4）哪些公司使用微服务架构

大部分大型网站系统如 Twitter、Netflix、Amazon 和 eBay 都已经从传统整体型架构（monolithic architecture）迁移到微服务架构。

5）微服务间通信

微服务通信方式取决于需求，可以使用 HTTP/REST、数据格式 JSON 或 protobuf，通信协议可以自由选择。此外，消息队列也是微服务架构中容器之间的常用通信方式。随着 REST Web 服务和 JSON 数据交换格式的流行，可以简单、快速地建立一个连接服务。微服务架构可以充分利用这些技术。

9.9.3　VirtualEnv 交付

Python 的 VirtualEnv 很好地解决了 Python 不同项目中对于库的依赖性，可以将不同的配置进行隔离而不会彼此影响。virtualenvwrapper 是一个建立在 VirtualEnv 上的工具，通过它可以方便创建、激活、管理、销毁虚拟环境；没有它的话进行以上操作将会相当麻烦。

VirtualEnv 仅仅是 Python 的版本和库管理器。VirtualEnv 带来的附加收益是将一个工程封装在了一个文件夹中。许多时候可以将该文件夹打包后进行交付，而不是通过一系列脚本来安装某个工程所需要的依赖项目和编译 C 扩展。

不过生产环境和开发环境还是有许多配置项目不同，需要采用后面提到的 Python 运维工具进行配置。

9.10　服务器运维

近来，Python 在服务器运维（DevOps）领域成为继 Bash shell 和 Perl 之后的主流编程语言。许多运维软件包都以 Python 为首选语言。这主要发生在以 IaaS/CaaS 为基础的设计中，因为 PaaS 的一般运维由供应商来做，运维工程师能够做的事情被限制在较小的权限内。不过即使如此，

服务器相关运维减少了，业务相关运维也依然不可缺少。

9.10.1 Linux 定时任务

Linux 中最常见的后台运维方式是 crontab，即定时运行的任务（Job）。这些任务由 crond（cron daemon）按照调度时间去运行，精确到分钟。其相当于时间触发的独立进程。这些任务和服务器进程之间可以通过数据库或者消息队列服务进行进程间通信，比如 SQL 数据库的某个字段或 Redis 的发布/订阅模型进行交互。

Linux contab 的常见任务调度可以大致分为系统任务调度和用户任务调度。系统任务调度包括写缓存到磁盘、日志清理等，用户任务调度有用户数据备份、定时邮件提醒等。

9.10.1.1 crontab 文件

crontab 文件的格式：

```
minute hour day month weekday username command
minute：分，值为 0-59
hour：小时，值为 1-23
day：天，值为 1-31
month：月，值为 1-12
weekday：星期，值为 0-6（0 代表星期天，1 代表星期一，依此类推）
username：要执行程序的用户，如：root
command：要执行的程序路径（绝对路径），例如：/root/crons/wipelog.sh
```

比较典型的调度示例如下：

```
0 6 * * * root /root/crons/wipelog.sh    每天 6:00 运行
35 5 * * 6 root /root/crons/wipelog.sh    每周六 5:35 执行
30 10-23/2 * * * root /root/crons/wipelog.sh    每天 10：30 起，每隔 2 小时执行
```

复杂的调度可以通过分解进程实现，更多例子请参阅本章延伸阅读部分的相关资源。crontab 的运行使用绝对路径，所以其调用的 shell 脚本也必须使用绝对路径。这降低了它的灵活性。虽然使用绝对路径有些不灵活，但却相对安全。解决方式是在被调用的 Bash shell 脚本中对路径做一些判断。

9.10.1.2 Plan

国内某位开源作者提供了一个 Python 包 Plan，简化了 crontab 的编写。可采用 pip 或源码安装。假设我们的路径中有一个 Bash shell 脚本。

wipelog.sh：

```
#!/bin/bash
mv twistd.log.* ./log/
```

很简单，就是将所有 log 文件清除到当前路径的 log 子目录中去。

plan_demo.py：

```
#!/usr/bin/env python
#/root/plan_lab/schedule.py
#
from plan import Plan

cron = Plan()

cron.command('/root/HybridDB_epic_socket/wipelog.sh', every='1.day', at='12:00')

if __name__ == '__main__':
    cron.run()
```

该脚本要求每日 12:00 正午执行一次 wipelog.sh。运行此 Python 脚本后，会显示对应的 crontab 任务。

```
# Begin Plan generated jobs for: main
0 12 * * * /root/HybridDB_epic_socket/wipelog.sh
# End Plan generated jobs for: main
```

如果没有问题，则可以将 plan_demo.py 中的 cron.run()改为 cron.write()。Python Plan 一共支持四种运行类型：

- cron.run('check')，默认值，在终端打印检查 crontab 语法，用于检查。
- cron.run('write')，用 Plan 创建 crontab 文件，替换现有 crontab 文件。
- cron.run('update')，在 crontab 文件中寻找 Plan 对象并进行局部更新。
- cron.run('clear')，在 crontab 文件中寻找并删除 Plan 对象。

1．Plan 参数

Plan 对象包括：

- Name，名称；
- User，用户；
- Environment Variable，环境变量；
- Bootstrap，自举；
- Pattern 模式。

planJob 类型包括：

- Command，普通可执行文件；
- Script，Python 脚本；
- Module Job，Python 模块；
- Raw Job，crontab 原生 Job。

若干 Plan 示例如下：

```
cron.command('cmd', every='1.day')
cron.script('script.py', path='/script_python/', every='1.month')
cron.module('calendar', every='feburary', at='day.3')

cron.command('top',every='4.hour',output=dict(stdout='/tmp/top_stdout.log',\
    stderr='/tmp/top_stderr.log'))
cron.script('script.py',every='1.day',path='/web/yourproject/scripts',\
    environment={'YOURAPP_ENV':'production'})
```

2．Every 参数

```
[1-60].minute
[1-24].hour
[1-31].day
[1-12].month
jan feb mar apr may jun jul aug sep oct nov dec
and all of those full month names(case insensitive)
sunday, monday, tuesday, wednesday, thursday, friday, saturday
weekday, weekend (case insensitive)
[1].year
"yearly"
# Run once a year at midnight on the morning of January 1"monthly"
# Run once a month at midnight on the morning of the first day# of the month"weekly"
# Run once a week at midnight on Sunday morning"daily"
# Run once a day at midnight"hourly"
# Run once an hour at the beginning of the hour"reboot"
# Run at startup
```

3．At 参数

直接对应于 crontab 的时间匹配模式参数。

```
minute.[0-59]
hour.[0-23]
hour:minute
day.[1-31]
sunday, monday, tuesday, wednesday, thursday, friday, saturday
weekday, weekend (case insensitive)
job=Job('onejob',every='1.day',at='hour.12 minute.15 minute.45')
# or even better
job=Job('onejob',every='1.day',at='12:15 12:45')
```

9.10.1.3　定时任务调试

许多情况下，需要打开 crontab log 跟踪调试 crontab job。crontab log 服务默认情况是关闭的。因为某些 crontab 执行频率比较高，会占用系统资源，所以调试完毕后，记得关闭 log 服务。

修改 rsyslog：

```
sudo vim /etc/rsyslog.d/50-default.conf
cron.*              /var/log/cron.log #将 cron 前面的注释符去掉
```

重启 rsyslog：

```
sudo  service rsyslog  restart
```

查看 crontab 日志：

```
cat  /var/log/cron.log
```

```
May 13 11:15:01 iZ2573cw0yvZ CRON[32075]: (root) CMD (command -v debian-sa1 > /dev/n
ull && debian-sa1 1 1)
May 13 11:17:01 iZ2573cw0yvZ CRON[32079]: (root) CMD (   cd / && run-parts --report /
etc/cron.hourly)
```

如果在日志文件中出现“No MTA installed, discarding output”，那么提示需要安装电邮服务。

crontab 执行脚本时不会直接输出错误信息，而会以邮件的形式发送到你的邮箱里。这时候就需要使用邮件服务器了。如果没有安装邮件服务器，它就会报错。如果是测试，可以用下面的办法来解决：在每条定时脚本后面加入：

```
>/dev/null2>&1
```

就可以解决“No MTA installed, discarding output”的报警。

从笔者个人的经验来看，在 crontab 中使用 shell 命令，调试起来很麻烦；使用 Python 脚本制作 crontab job，更加符合笔者的习惯，而且由于在 Python 程序中可以判断路径，因此可以部分规避相对路径和绝对路径的麻烦。

9.10.1.4　APScheduler

APScheduler（Advanced Python Scheduler）脱胎于 Java Quartz。它实现了类似于 crontab 的进程间调度器。但和 Plan 不同，APScheduler 并不操纵系统定时服务如 Linux crontab 和 Windows task scheduler。APScheduler 本身并不是守护程序，而是立足在当前应用中提供一个方法，产生一个独立进程来延时调度或者周期性调度某些 Python 任务。

纯 Python 实现的 APScheduler 是一个跨平台的选择，这一点很有吸引力。APScheduler 内置三种调度系统。

- Cron 风格：带可选的启动/停止时间；
- 间隔执行：每隔一段时间运行一次，带可选的启动/停止时间；
- 延时执行：在预设的日期和时间，仅运行一次。

开发者可以混用不同的调用系统，并采用自己喜欢的后端来存储任务，包括：

- 内存；
- SQLAlchemy，支持任何 RDBMS；
- MongoDB；
- Redis。

APScheduler 还能够集成多个常见的 Python 框架：

- asyncio（PEP 3156）；
- gevent；
- Tornado；
- Twisted；
- Qt（PyQt 或 PySide）。

APScheduler 很小巧，不到 80KB，可以采用 pip 安装。

9.10.2 常见的定时任务

现在我们罗列一下日常运维中可能会遇到的定时任务。

9.10.2.1 数据压缩与归档

在笔者设计的物联网工程中，客户定义了数据交换文件，采用 XML 格式，用于服务器、PC 客户端和手机客户端之间的数据交换，XML 文件的最大问题在于文件尺寸太大。单一设备每天产生的 XML 记录文件就有 10MB，1000 台设备就会产生 10GB，一个月就会产生 300GB 出网流量。

如果每天从服务器下载 XML 格式的文件，无论是包月还是按需计价的流量方案，都不合理。这样既占用带宽堵塞、正常服务，又产生巨额费用。此类归档服务必须采用压缩传输。经过实验，在 Linux 中采用 gzip 压缩普通的 XML/JSON 文件，可以有 10～30 倍的压缩比。这可以释放大量计算能力和存储资源，降低了费用。

与此同时，产生这些归档文件本身需要对数据库进行检索和删除操作，在 IOPS 不足的数据库上操作会形成慢 SQL 操作。所以，笔者将此类文件安排在系统闲时操作。这也是需要定时操作的原因。

9.10.2.2 数据收集与统计

在日常运维时需要进行统计数据的收集。主要的设计任务就是查询数据库，然后将根据需求对所有设备的连接数量、IP 地址、地理位置、登录时段（忙时与闲时）以及所收集物理量的峰值谷值、均值、中值和分布进行统计学分析和二次数据收集，并记录在新的数据库和归档文件中。

这些统计与服务器端数据收集任务需要单独设计模块，并交由 Plan/APScheduler 来调度。这方面的需求复杂度甚至比物联网接入服务还要高，因为这部分实际上已经涉及业务逻辑。

以上 Python 包主要用于小规模的运维任务。最终系统会变得越来越复杂，并运行在分布式集群中。开发者需要一些更加专业化、自动化的运维框架，以便能够及时配置和管理按需分配的每个服务器实例。

9.10.3　系统监控

作为运维工程师，必须要经常了解服务器中各种资源的使用情况，并在服务器进入临界情况前做处理。一旦服务器进程因太占用资源而被操作系统杀掉，就需要手动或者自动重启进程。这些都依赖于一些监控软件包，而且监控进程必须独立于服务器进程，并作为系统服务在后台运行。

9.10.3.1　psutil

psutil 的全称为 Python system and process utitlies，它类似于 Linux 中的 ps 工具。其可以让运维程序来了解系统的 CPU 占比、存储器占比、网络使用状态。在此基础上，运维团队可以实时监控系统、获得报警信息、增减计算资源。psutil 采用 pip 安装。

```
#!/usr/bin/env python

import psutil
import datetime
from subprocess import PIPE

import sys

'''
python wrapper for system admin tools including:
ps|top|lso|nice|netstat|ifconfig|who|df|kill|free
ionice|iostat|iotop|uptime|pidof|tty|taskset|pmap & etc
'''

print "CPU related"
print psutil.cpu_times()
print psutil.cpu_times().user
print psutil.cpu_count()
print psutil.cpu_count(logical=False)

print "MEM related"
mem = psutil.virtual_memory()
print mem.total
print mem.free
```

```
print "DISK related"
print psutil.swap_memory()

print psutil.disk_io_counters()
print psutil.disk_io_counters(perdisk=True)
print psutil.disk_partitions()
print psutil.disk_usage('/')

print "NET related"
print psutil.net_io_counters()
print psutil.net_io_counters(pernic=True)

print "USER & Time"
print psutil.users()

print psutil.boot_time()
print datetime.datetime.fromtimestamp(psutil.boot_time()).\
    strftime('%Y-%m-%d %H:%M%S')

print "PID related"
print psutil.pids()

for i in psutil.pids():
    p = psutil.Process(i)
    print p.name()
    print p.exe()
    print p.cwd()
    print p.status()
    print p.create_time()
    print p.uids()
    print p.gids()
    print p.cpu_times()
    print p.cpu_affinity()
    print p.memory_percent()
    print p.memory_info()
    print p.io_counters()
    print p.connections()
    print p.num_threads()

p = psutil.Popen(["/usr/bin/python", "-c", "print('hello')"], stdout=PIPE)

p.name()

p.username()
p.kill()
sys.exit()
```

以上例子充分说明了 psutil 的使用方法。最后一段说明，psutil 还可以根据所获信息启动进程。这可以用于重启服务器、发送电邮等进程。

9.10.3.2　Python 与系统服务

虽然 Python 已经提供了许多手段和服务器封装，但是某些情况下仍需要单独设计系统服务。这些服务既不是标准的服务器（Web/FTP/SSH/FTP/Telnet），也不是中间件，而有可能是某些进程的守护程序（daemon）。这些程序不需要维持多个长连接，也不是按时运行，而是监控某种异步事件。比如进程被系统杀掉，或者收到某个报警等，并做出相应的处理。

daemon 可以采用任意语言（Perl/Python 或其他）编写，只要设计定义成为系统服务即可。Python 配合 Linux 可以很好地完成此类任务。在 Python 中的实现方式如下：python-daemon，带定制 shell 调用的 subprocess；还可以使用 os.fork()等实现。

在 7.4.10 节中，我们了解了 Linux 中的系统服务演进，从 System V init，到 upstart，再到 systemd，以及利用 Windows 中的 sc.exe 产生系统服务的方法。在服务器中可以采用同样的方式实现系统服务。

9.10.4　集成化运维软件

除了这些小型的扩展包，Python 还有许多集成化的运维软件，可以大大简化运维的工作量。这也是 Python 之所以在运维应用中得到大规模普及的原因。

常见的 Linux 运维有如下场景：

- 操作系统和软件的安装、配置、初始化等，可以使用 Puppet、Chef、CFEngine、Ansible、SaltStack；
- 自动执行任务，比如定期备份、清除日志等，可以使用 Fabric、Ansible、SaltStack；
- 手动执行任务，比如部署应用、升级、重启、检查和校验文件系统、增加用户等，可以使用 Rake、Fabric、Ansible、SaltStack。

9.10.4.1　Fabric

Fabric 是一个 Python 2.7 库和命令行工具，用于利用 SSH 远程登录，并进行应用部署和系统管理任务。Fabric 可使用 pip 安装，Fabric 依赖于 Paramiko、pycrypto、ecdsa 等与 SSH 相关的包。

9.10.4.2　Puppet

服务器的初始配置由 Puppet 配置管理工具来管理。单台服务器实例上线后由 Puppet 完成初始化和配置等一系列工作，如静态 IP 分配、DNS、NFS/SAN 挂载、LDAP/Kerberos 登录、安全加固配置、内核参数优化及防火墙规则等。

初始化完成并运行一段时间后，也会有一些需要自动和手动操作的任务，比如升级、重启、备份等，这时候再使用 Fabric 来批量执行这些临时任务。

9.10.4.3 SaltStack

SaltStack 是一个异构平台基础设置管理工具，其使用轻量级消息队列 ZeroMQ，并采用 Python 写成批量管理工具。其拥有强大的远程执行命令引擎和配置管理系统。SaltStack 的依赖项比较多。

上面提到的 Puppet 和 Fabric 其实是两个不同性质的工具。而这两个工具可以由单一工具 SaltStack（或 AnsibleWorks）完成，以减少学习成本、人力成本和时间成本。由于 SaltStack 采用 ZeroMQ 消息队列进行通信，而且采用 Python 编写，因此与 Puppet/Chef 等 Ruby 工具相比，其速度更快，值得推荐。

9.10.4.4 Ansible

Ansible 是自动化运维工具，基于 Python 开发。其糅合了许多老牌运维工具的优点，实现与了批量操作系统配置、批量程序部署、批量运行命令等功能。Ansible 由 Paramiko（SSH 并发连接）、PyYAML 和 Jinja2 三个关键模块组成。

Ansible 提供商业许可证版本 Ansible Tower。与 SaltStack 相比，其对于运维人员的要求更低。

若欲了解更多 Python 运维工具信息，可以参阅 *Python for UNIX and Linux System Adminstration* 一书。

9.11 物联网系统设计实践

在为客户开发多种物联网应用系统之后，笔者觉得有必要总结一下整个过程中得到的经验与教训，并与读者分享。

9.11.1 服务器端需求分析

无论提出实施物联网的团队来自组织外部还是组织内部，开发团队都需要抛弃成见，虚心聆听客户的需求。主要目的如下。

- 了解客户设备联网的目的和核心价值。了解设备联网需要解决的痛点和附加值，这对数据分析设计有意义。
- 了解客户所在行业的生态和趋势。尤其是要了解可扩展性和应用融合的趋势，这对架构和接口设计有一定意义。
- 定义客户短期可实施目标，打消客户不切实际的想法。

在需求分析实务中，需要收集的需求信息如下。

- 设备数量：产品生命周期，样机、小批量、中等批量和长期数量，开发周期与固件升级；
- 数据特性：数据上传/下发和转发端口的协议、格式、频率、带宽、容量和实时性；
- 数据存储：数据持久性、周期性、是否需要归档和大数据数据库，以及查询条件和实时性特性；
- 统计分析：数据统计、分析以及其他处理；
- 触发条件：对于电邮、短信、报警等的触发条件；
- 认证授权：企业和第三方的 SSO/Auth 需求，管理授权策略；
- 应用融合：与第三方和 CRM/ERP 的系统融合需求；
- Web 前端：IE 兼容要求，响应式 HTML 设计，数据实时绘图、历史绘图；
- APP 接入：是否有 APP 接入需求，支持哪些终端；
- 附加需求：其他技术和商务需求。

在需求分析阶段，客户会将对于所处行业的理解和展望向开发者和盘托出。笔者有时候非常钦佩这些企业家们，他们往往会有很强的前瞻性和危机感，发散性思维远超我们的想象。其中，一些先行者们还有着整合本行业和颠覆其他行业的雄心。

在笔者的一些工程经历中，谈判需求一开始，就容易陷入一些细节无法自拔。客户的“脑洞”都很大，他们会提出一大堆古怪的需求来：比如，和组织机构有关的角色和权限管理，以及非标准通信协议、非标准的时间和时区定义、违反最佳实践的需求、违反商业道德的需求等。提供一个相对现成的方案可以避免落入外行指挥内行的陷阱；对于客户的某些违反最佳实践和商业道德的需求，要说服客户放弃。

企业的任何宏伟计划均必须先走出第一步：设备联网。没有哪一种架构可以同时满足最小规模到超大规模的需求。开发计划的实现目标必须落实在 1~2 年内的连接数规模。一定要打消企业家们一步到位的想法。否则，系统需求分析本身会变得格外漫长，无法落地。

笔者经常为企业引用一个创业团队的例子：一位来自著名互联网企业的工程师出任新创企业的 CTO，在业务发展之初其就设计构建了一个超大规模架构，结果浪费了资源和时间，最后不得不推倒重来，设计更加小型化的架构以适应现有业务规模。关于如何选择和设计合适的系统架构，可以寻找此类参考书来看。例如：

- 《大型网站技术架构：核心原理与案例分析》（作者：李智慧，电子工业出版社出版）；
- 《程序员必读之软件架构》（作者：Simon Brown，人民邮电出版社翻译出版）；
- 《恰如其分的软件架构》（作者：George Fairbanks，华中科技大学出版社翻译出版）。

现阶段有许多传统企业希望通过升级物联网系统能够**一次性**解决所有的企业升级问题，实际上这是一种不太现实的期待，我们必须打消企业的这种期待。因为这种需求必然追求大而全

的模式，与互联网“精益创业”的 MVP 开发趋势背道而驰。

互联网和物联网发展得太快，尤其是在如何进行应用融合这一方面，企业自己的想法往往也是一日三变。比较实际的方式是将物联网设备云单独运行，与业务运营解耦；而应用、业务和第三方扩展通过单独的业务服务器，通过 Web API 和认证授权来访问设备云数据。这种方式首先解决了设备联网，其次解决了业务单独发展的目的。分而治之，构建速度快，迭代速度也快。

总的一句话：物联网设备云负责“物”，互联网各类应用云负责“用户和业务”。同时，物联网中设备与用户的数据分离也部分解决了隐私问题。

9.11.2 确定设备接入方式

物联网设备端和服务器端之间虽然使用标准 TCP/IP 协议栈，但是 WSN 组网和联网协议的选择却是五花八门的。具体选择请参阅第 5 章。更多的设备供应商连接中往往偏向采用自定义组网和联网协议，这些协议大都从原有串口和工业总线上转换而来，有许多底层协议和不尽合理的帧结构。总的原则是分段确定设备接入方式。

- 网络拓扑：选择蜂窝数据模块直连或通过网关连接服务器；
- 组网技术：从移动蜂窝数据模块、NB-IoT、LPWA、WSN、BLE 或 WiFi 中进行选择；
- 联网协议：选择基于套接字实现定制协议或是直接采用标准化协议。

在联网协议中，还要确定逻辑分层细节。

- 传输层：TCP 长连接、短连接或 UDP 连接；
- 应用层：MQTT、CoAP、HTTP 或其他自定义协议；
- 序列化：XML、JSON、msgpack、protobuf 序列化协议或自定义协议。

开发者应尽可能说服客户逐渐向各类标准化协议转化。但由于客户的历史遗留包袱，因此往往不得不支持客户的私有协议。这是一个需要彼此妥协的过程。以 Twisted 为例，其作为 TCP/IP 软件包的集大成者，已经支持了大量的应用层协议。可以基于 Twisted 支持私有协议定制。

前面提到过，TCP 是基于数据流的协议，容易出现粘包和半包现象，继而导致解析失败的现象。必须在接收底层进行拼合，然后发给高层解析程序处理。关于这一点，Twisted 的创始人 Glyph Lefkowitz 专门在 Stackoverflow 上回复了笔者的提问，提供了参考设计。此外，Cyclone 框架的 WebSocket 中对于粘包的处理也对笔者有一定启发。同时，也可以采用 RTS/CTS 软件流控方式、TCP 短连接和 UDP 方式来回避此问题。可以基于 Twisted 支持私有协议定制，但一定要清晰地向用户表达私有协议的局限性：安全受限、扩展困难、升级困难以及规模受限。

如果采用标准化的物联网协议，那么基本上会在 CoAP、MQTT 两者中做个选择。在 9.4 节中，笔者已经介绍过了这两种协议的异同。现在总结一下：

- CoAP 基于 UDP，适合组网，支持 D-TLS 安全标准；
- MQTT 基于 TCP 长连接，适合联网，支持 TLS 安全标准，如物联网网关联网；
- MQTT/HTTP/TLS 对于设备端计算能力有较高要求，可以选用自带 MQTT/TLS 能力的 MODEM。

物联网，尤其是国内的物联网应用对于安全是非常忽视的。大多数蜂窝数据/Wi-Fi/BLE 模块并没有支持 TLS/MQTT，而依靠普通开发者基于 TCP 实现，水平参差不齐。可以这么说，大多数系统安全性甚至远远不如 Mifare RFID。所以即使物联网设备计算资源有限，无法支持 TCP/IP+MQTT+TLS，也请参考标准、标准草案和开源代码，实现双向认证、AES 加密传输，并尽可能参考逻辑分层设计联网协议。

综上所述，设备连接的选择存在着很多依赖性，需要和客户仔细商讨。

9.11.3　物联网的实时要求

物联网相对互联网来说，有个隐含的要求：**实时性**。实际上，在笔者承接物联网项目时，发现这是许多客户的需求。数据采集后，实现实时绘图和实时数据分析是其主要的功能需求。

一些客户甚至提出了想要通过物联网进行实时远程遥控机械手的要求。在自动控制原理中，系统环路的延时对于系统的传递函数频率特性影响很大。虽然固网宽带和 4G 移动宽带发展神速，但是公众网络的传输天然具有不稳定性，这使得这种方式变得不可控。千万不要把自动控制的环路放大到整个网络上。基于公网和远程控制与基于局域网的远程控制是不同的，物联网系统设计需要正确地切割实时控制和网络传输任务，并定义彼此之间的接口。

实时性需求是对系统架构的一大挑战。所谓实时性，是数据从采集、流经公众网络、到达云服务器，经过转换、存储、转发之后，再转发到另一台设备或者客户端的整个过程所需要的时间。系统架构需要从多个方面进行优化才能够满足整体实时性需求，主要需要在设备接入、数据持久层方面优化。其中，数据持久层的优化，包括数据库的选择、建模和优化更加重要。

9.11.4　EPIC IoT 设备服务器

根据既往的开发经验，笔者基于 Twisted/Cyclone 和其他开源组件，提供了一套可扩展的 Python 物联网服务器整体设计。方案全称为 Extensible Easy Python IoT/IoE Cloud，简称为 EPIC。这里计划以问答式的工程向导，让用户选择联网方式与协议，并构建最简单的物联网应用。

EPIC 的最初目标是为中小规模用户提供一个现成的多合一平台。但随着研发逐渐展开，开始导入分布式设计。首先将设备接入管理独立构成设备服务器，并通过 Web API 将设备抽象化，与高层应用有关的部分单独设计一个服务器，通过 Web API 访问设备服务器，继而访问设备。

这套方案的特点如下：

- 采用 Python 编写，可使用 CPython/Jython/PyPy 运行时；
- 所有组件可以基于 IaaS 开发，并支持以 Web PaaS + IoT PaaS 方式部署；
- 由独立设备接入服务器（设备云）和资产管理应用服务器（应用云）构成；
- 设备云基于 Twisted/Cyclone 异步框架构成；
- 应用云基于 Flask 同步框架构成，并使用 gevent 实现性能升级；
- 设备云与应用云之间采用 TLS+REST API 连接；
- 设备云与应用云对第三方和 APP 提供 TLS+REST API 连接；
- 提供开源的套接字接入协议和 ARM mbed 参考设计。

缺点如下：

CPython 的运行速度不如其他面向性能的语言。连接数较多时，CPU 占比处于高位。单台服务器并发数不高。设备接入部分基于 Twisted，属于纯 Python 代码，笔者现在找到了若干加速的方法：

- 采用 Ngnix 负载均衡，后接工作服务器若干，支持 UDP/TCP/HTTP 端口；
- 采用 IoT PaaS/mosquitto，多使用高并发的 MQTT 端口；
- 采用 PyPy 替代 CPython；
- 采用 Golang 替代 CPython，或实现混合编程；
- 采用 Cython/libuv/pyuv 替代 CPython/Twisted 自带的 epoll/select 底层。

前两种方法为使用负载均衡和代理服务器来提高系统的高并发连接数量。而后三种方式为替换了 CPython 和 Twisted 的底层实现，可以降低单台服务器的 CPU 占比，继而增加单台服务器计入数量，减少服务器实例。根据 pyuv 作者 Saúl Ibarra Corretgé 的测试，加速后的服务器运行速度与 Golang 在同一数量级。

9.11.4.1 应用服务器

设备服务器负责设备连接和管理，但是组织结构相关的业务逻辑、授权、认证的设计则需要在业务应用服务器中实施。设计方法如下。

- 连接方式：两个服务器之间采用 REST API 或消息队列进行整合。
- 资源管理：每台设备都是设备服务器的“资源”。
- 读/写密钥：每种资源有若干密钥对，分别对应读/写密钥，每个密钥有对应的 API-ID 和 API-key。
- 密钥分发：API-key 和 API-secret 在设备服务器中产生和管理，但是业务应用服务器也可以进行密钥分发。

作为第三方应用，如果需要访问设备云数据，则首先需要从设备或业务服务器处申请 API-key 和 API-secret，然后利用这对密钥访问设备服务器中的设备数据。

业务相关应用服务器与普通 Web 服务器设计需求、流程和方式是一致的。参考 Flask/Django/Tornado 的设计，配合 Bootstrap 前端 UI 就可以设计出一个成熟的网站来。笔者推荐 Flask+gevent 的框架，其灵活性更大一些，包括 RBAC 部分都可以参阅 Flask 参考书，使用 flask-rbac 进行实施。

9.11.4.2　压力测试

针对 HTTP 服务器，许多开发者采用 AB（Apache Benchmark）做压力测试。这是 Apache 提供的 C 语言编写的压测工具。但是 AB 压测有卡顿的现象，这可能与某内部的程序实现有关联。

与虚拟设备不同，压测工具的性能需要超越服务器性能才能够测试出服务器性能阈值。这一点让笔者颇费思量：

- 考虑到 CPython 多进程受到 GIL 的限制，笔者设计了多进程的 Python 压测程序。
- 多进程受限于 CPU 核数，也不是理想的压测环境。
- 采用 Jython 回避 GIL，实现多线程压测程序，但是性能也不够理想。
- 采用远程压力测试，虽然是网络真实环境，但却引入了额外系统延时，测试结果不够纯粹。
- 压力测试还需要分为空载压测和全业务压测。

在其他语言中，Ruby 也有 GIL 问题，Lua 也不是真正的多线程，而 Scala 和 Erlang 的学习曲线太长。可以使用 C++/Go 做压力测试。部分云计算服务商如阿里云，提供基于分布式集群，通过内网进行压力测试，这可以比较真实地反映压力测试的结果。

目前 EPIC 单机版在阿里云上已经做过压力测试。测试结果得出了 CPython 的 CPU 占比较高，MySQL 有性能瓶颈的结论。

9.11.5　EPIC 架构优化

不同的连接数对应不同的系统架构，从千、万、十万到百万甚至更多数量级，每一次升级，整体架构都需要来一次彻底的更新。

总之，架构优化是一件长期贯穿整个项目生命周期的事情，所以不要奢望一种架构可以覆盖所有的需求。如果架构无法扩展，则无法满足业务快速扩张的需求；反之，业务太小时提前规划超大规模的架构也是枉然。DevOps 团队在系统的整个生命周期中必须紧跟业务的发展进行持续改进和扩充。

由于 EPIC 是一种通用的网络应用结构，因此笔者规划了不同的版本。针对不同的版本，EPIC 的架构在开发中进行持续优化。

9.11.5.1　单机版本

此类平台以标准服务器操作系统为主，是 EPIC 的开发基础。单机版本可以用作系统开发，但是不太推荐用于生产环境。无论是家用 PC、笔记本、虚拟机或单台服务器，都可以归类于这

种平台。单机版本也可以运行于单一云服务器实例中。

这里以阿里云的标准（低）配置为例：1 核 ECS，1GB RAM，20GB 系统盘，安装 Ubuntu Server 32 位。由于笔者做的压力测试是针对全业务运行的，没有单独对套接字服务器进行压测，因此无法给出空载压测数据。笔者在这个系统里安装了多个 Twisted 服务器、Cyclone Web 服务器、Redis 服务器。在业务整合后，压力测试的测试结果为，1MB 带宽下可以支持 1000 多台设备长连接。只是 CPU 占比较高，所以需要采用 PyPy 做加速。

单核配置适用于大多数中小规模的设备数量。高配服务器（Scale Up）可以支持到 10000 台设备。当然多核处理器本身就需要构建反向代理服务器做负载均衡，同时将单机 MySQL 搬到 RDS，将单机 Redis 搬到 KVStore 服务器上。

实际上，10000 个用户数在商业上已经是一道坎了，这时就不太推荐使用单机服务器了。毕竟单点故障的概率和风险很大。如果你的产品已经卖出了 1 万套，那么就应该考虑采用其他技术的云服务进行系统升级，以降低风险。

1．MySQL 瓶颈

EPIC 最初的设计使用了类似 LAMP 的方式，只不过没有使用 Apache 和 PHP，而是直接使用了 Python。但是这无法满足物联网的实时性要求，瓶颈不在于 Python，而在于 MySQL。

LAMP 之所以在互联网中得到大规模普及，一方面是因为开源的力量，另一方面是因为它适应了互联网的数据流模型。一般的面对浏览器的使用场景，数据下载流量远大于数据上传流量。时序数据的数据特性和数据库检索需求与传统互联网应用存在较大差异。

而物联网应用的主要功能之一是数据收集。无论这个数据量有多小，如果有一定的设备量，则日积月累的数据会很快累积到一个惊人的数量级。一些应用如车联网和医疗设备，对于数据采样频率有较高要求，每日单台数据都能够到达 1GB。笔者自己的某个项目 2 天内 3 台样机采集的实时采集数据就累积到了 600MB。平均每台设备每天累计数据为 100MB。如果按照 10000 台设备计算，每天的入网流量如下：

```
0.1GB×10000 = 1TB
```

数据读/写对于 MySQL 和其他任何 SQL 来说时间不算长，为毫秒级别。但如果客户要求实时提取数据，那么问题就来了。提取数据的脚本会针对特定设备 ID，采用时间过滤的方式，查询提取数据。这样，在 1TB 数据点中检索的速度就很慢了。所以，在单实例服务器中，数据库定时归档及删除/归档过期数据是必需的，这可以减少检索的数据集规模，减少检索返回时间。

熟悉 SQL 数据库的 DBA 都知道，删除数据需要锁定数据库，直到操作完成。而删除 1TB 的数据本身会造成锁表时间从几秒到几十秒不等。这期间任何对该表的操作都被禁止。这就变成了一个死循环：

- 不删除数据，数据表检索会越来越慢。

- 删除数据，数据表会被锁定几十秒。

为了解决这个问题，笔者想了许多方法尝试解决：数据库分片，表格切换，等等。笔者最后终于认识到，至少 MySQL 不适用于这种实时数据存储，即使采用内存表也会发生表锁。读者必须抛弃既有的 Web 架构，采用更合适这种场景的数据库和架构。

2．Redis/MySQL

笔者通过一段时间的摸索和多方面了解，发现两个事实：

- 实时数据，目的是为了分享，或许不用持久保存。
- 数据模式，应该被称为缓存（Cache），主要保存在 RAM 中。

通过对比 Memcached 和 Redis，笔者选择 Redis 作为实时数据的缓存数据库；并采用数据定时消失的方式，解决了需要定时删除数据点的问题。不仅数据插入和检索速度惊人地快，而且规模可控。对于需要长期保存而数据量比较小的数据，依然保存在 SQL 数据库中；更大的数据保存于 MongoDB 中。

就此，形成了 Redis+SQL 数据库的梯度存储结构，在单机实例中也可以处理“海量”数据的设计了。其处理上限受限于 CPU 占比和 RAM 容量。EPIC 单机实例的物联网服务器架构如图 9-14 所示。

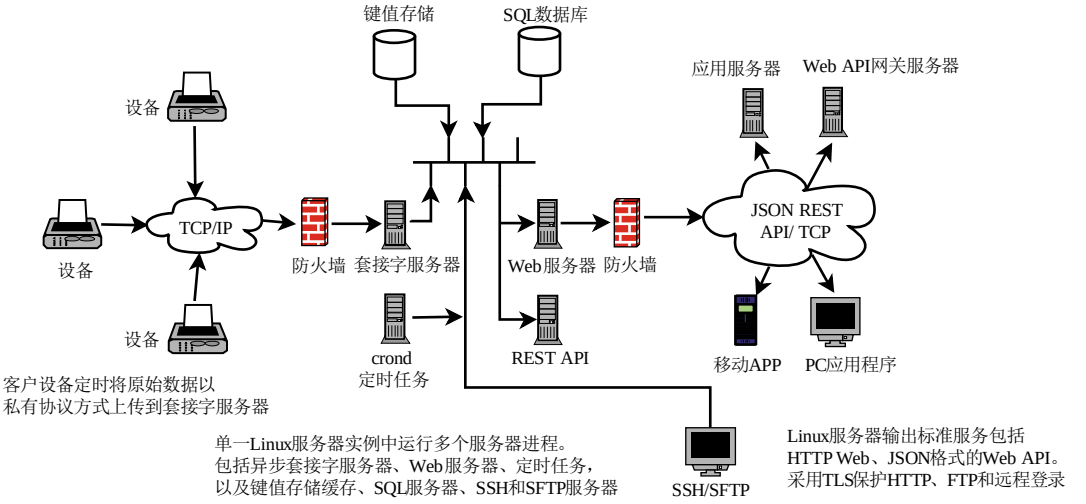


图 9-14　EPIC 单机版物联网服务器系统架构图

这种缓存数据除了可以实现实时绘图，其实还可以支持实时监控和数据分析。但是目前的监控都太简单：超过某个阈值时就报警。许多时序信号可以采用傅里叶变换/小波变换进行实时分析。典型的例子如下：

- 通过音频频谱分析来了解肺部栓塞情况，所谓“痰音”是经验之谈，需要通过频谱分析进行量化；
- 通过测量电梯的振动和声音，可以知道设备运行情况和危险的不正常情况；
- 通过测量区域内大量手机的加速度计信号，可以知道地震情况。

这些都是计算密集型任务。通常，如果设备节点计算能力足够，则可以在本地进行分析。即进行所谓的边缘计算。但是许多设备仅仅具备传感器数据收集和传输能力，并不具备足够的计算能力，这就需要依靠云服务器来做。

物联网大多数是 I/O 密集型计算，其 KPI 是连接数和 CPU 占比。Redis 算存储密集型计算。傅里叶变换和小波变换是计算密集型计算。如果都进行实施，则物联网的架构就需要另外设计了。目前的架构无法实现对所有的设备节点进行实时流式计算，但针对选定设备做这些数学计算还是可行的。

由于 Redis 支持 Pub/Sub 模式，因此笔者觉得还有挖掘的潜力：通过该模式实现数据接收后，可通过 Redis 分发给数据库、实时数据分析服务、实时监控服务和第三方服务器，以减轻数据库检索的压力。

9.11.5.2 分布式版本

分布式版本依托云计算供应商的分布式 PaaS，如负载均衡、分布式数据库、TSDB 等各种手段，以及基于云服务器实例集群构建的数据库。其主要用于解决大规模设备接入情况下产生的系统性能瓶颈，并为大数据分析提供技术支持，挖掘数据“金矿”。

分布式版本适用于大规模部署，同时在线设备取决于投入的资金；并通过不断追加投资，购买更多资源来服务更多在线设备。

MQTT/Redis/TSDB

在分布式版本中支持四种连接方式：

- Ngnix + Twisted，用于 REST Web API；
- Ngnix + Twisted，用于 TCP socket server；
- MQTT；
- CoAP + Twisted。

各类消息队列是解决连接问题的优选方案，可用于解决物联网实时性和异步通信问题。

传统的互联网架构和存储架构，所有的数据必须流经数据库。而 MQTT 可通过消息队列总线转发给目标客户端，需要持久保存的数据可转发给后台数据库进行存储。Redis 也支持 Pub/Sub 模式，同样也作为消息队列，不过其缺乏 MQTT 协议支持。但 Redis 可作为进程间异步通信消息总线，比如每日的批量处理、触发电邮或者短信报警等。

这种基于消息队列的架构被称为微服务架构。一个服务器系统中需要许多服务，如归档、电邮、数据持久、报警推送等。如果都设计在同一个复合型服务器中，则进程间通信太过复杂，耦合度也太紧密，不利于维护升级。在微服务架构中，将每个服务设计成单一服务进程，通过消息队列耦合，可以单独升级某一服务而不影响其他服务。

在数据存储设计上，复用 Redis 做实时数据缓存，分布式 SQL 做短期数据存储，长期数据使用压缩归档服务。

针对更大连接规模、更灵活的运维，图 9-15 描绘的 EPIC 分布版物联网服务器系统架构将是满足物联网系统的可扩展需求的设计之一。

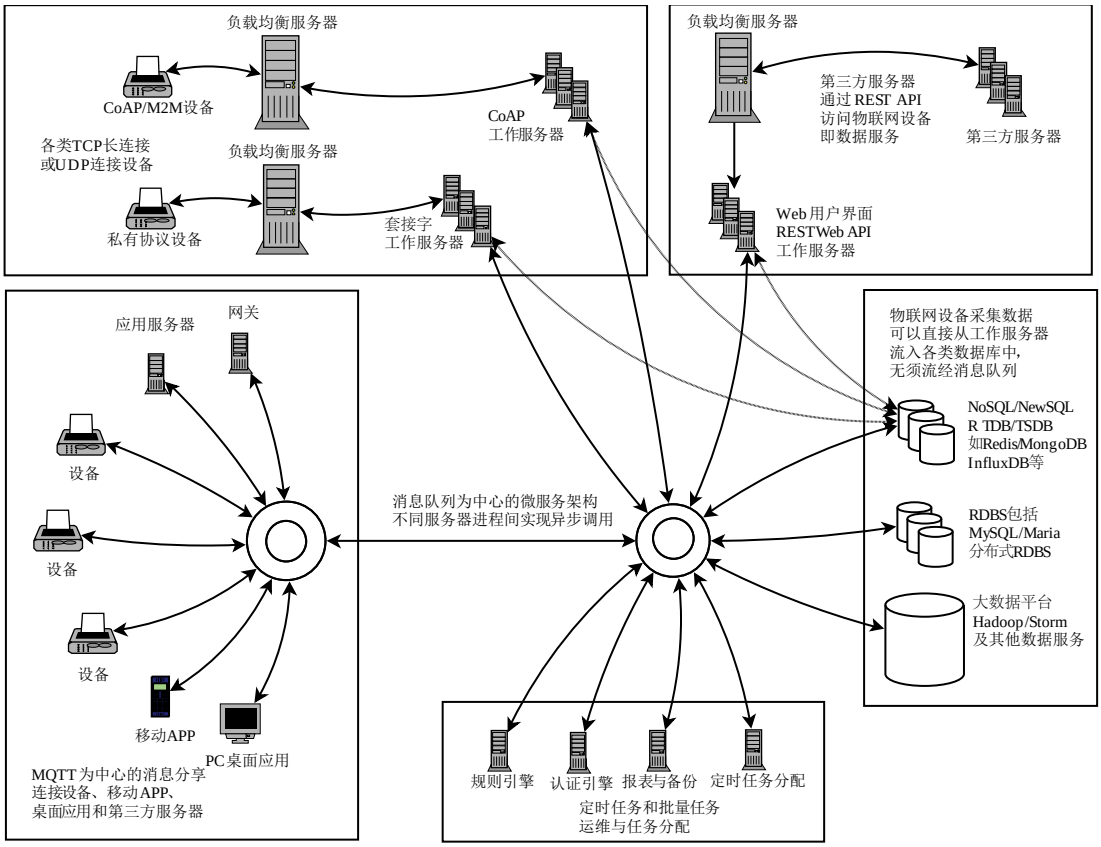


图 9-15　EPIC 分布版物联网服务器系统架构图

- 设备接入：MQTT 将成为标准接入方式，使用 IoT PaaS 标准 MQTT 接入服务。设备、APP、第三方服务器都可以接入分享数据。
- 设备接入：传统设备接入方式，支持 socket server、Web Server 和 CoAP Proxy，采用

Ngnix 代理服务器做负载均衡。

- 消息总线：不同服务间采用消息队列总线耦合。
- 数据持久：采用 Redis/SQL/MongoDB 进行梯度存储，其中大数据部分可以使用 Lambda 架构。
- 时序数据库：评估各种实时时序数据库，目前使用 InfluxDB。
- 任务调度：crond 实现周期性任务调度；
- Web API：Web Server 实现 REST API、HTTP SSE 等。
- 数据安全：TLS/D-TLS 分别保护 HTTP/MQTT/CoAP。
- PaaS 化：尽量采用云计算供应商的 Web PaaS 和 IoT PaaS。
- CaaS 化：采用容器和开源组件实现 MQTT/CoAP/Redis/Ngnix/NoSQL/NewSQL/TSDB。
- 资源加速：采用 CDN 服务加速。
- 冗余备份：定时归档和分析。

在阿里云中，负载均衡需要分为四层和七层两种。其中四层就是提供到 TCP/UDP 传输层的负载均衡，类似于 Ngnix，可以支持 50 万个连接数量。七层提供到 HTTP/HTTPS 应用层的负载均衡，由 Tengine 负责。该项目曾经在淘宝、天猫商城中部署，其可以承担较大的连接数量，但是有一些限制。

在本架构中使用 Ngnix 作为负载均衡，并同时采用 MQTT 做设备接入。开源 MQTT 组件 mosquitto 支持 TLS 和 Auth 插件，也可以实现 2 万台设备接入，而 Ngnix 可以支持 50 万台设备接入。接入数量的基数为 52 万个。此外可以通过子域名、MQTT 前置负载均衡，以进一步扩展系统接入数量。

在分布式版本的时序数据库中，笔者主要评估了 InfluxDB 和 OpenTSDB。InfluxDB 比较方便，但是因其提供 Web 管理界面，所以我们需要留意安全相关配置。

9.11.5.3 迷你版本

EPIC 同样可以运行在嵌入式平台中。对于嵌入式设备，系统连接数量上也降低了要求，支持 10~200 个连接数就足够了。同时，在 Linux 平台中，通过 MicroPython/PyMite/CPython/Jython，Python 的运行环境也算完整，就连 Android 中也可以运行 Twisted。

EPIC 嵌入式版本可以支持 MQTT/CoAP/socket server 的接入方式，也可以支持 Redis 内存数据库。

但嵌入式系统与标准服务器之间的最大差别或许在存储介质上，这一点对数据库的选择产生了重要影响。

闪存友好型数据库

在嵌入式设备中往往使用闪存技术，这对数据更新次数其实是有限制的。这一点笔者深有

体会。一方面笔者来自半导体行业，对于固态存储器有一定了解；另外一方面，迄今为止笔者已经用坏了很多闪存卡了。所以，消费级的 SD/TF 卡实在无法承受长期大量数据的压力；而且操作系统和数据保存在同一闪存中，风险也较大。

所以，数据至少需要在 RAM 中缓存到一定数量再写盘，才可以减少对于闪存的损耗。挑选存储硬件和软件架构很重要。在这方面笔者有以下几点可以分享。

- 热点数据：可使用 Redis 做缓存。
- 记录数据：在 RAM 中累计数据后，可以一次性保存到闪存中，以减少写入次数。
- 采用 eMMC：板载 eMMC 质量好过外置 TF/microSD 卡。
- 采用 SSD：虽然厂家参差不齐，但至少内置的控制器拥有平衡写入负载的算法。
- 外部存储：考虑使用 NFS（网络文件系统）或者 USB 硬盘作为数据存储。

此外，虽然嵌入式架构 SBC 中现在也出现了 64GB ROM/4GB RAM 的高配控制板，但是计算能力仍无法与服务器相提并论。选择合适的组件或者对某些特性进行裁剪也是必需的：

- 套接字服务器：Twisted；
- Web 服务器：用 Klein 替代 Cyclone；
- MQTT：mosquitto；
- CoAP：局域网内物联网协议的首选；
- 消息队列：MQTT/Redis 二选一；
- 热点数据：Redis；
- 数据库：SQLite/MySQL Embedded 或内存数据库。

关键在于，尽量避免对于系统闪存的过度使用。

9.12　本章小结

由于服务器端涉及的环节比较多，包括了云计算模式对比、物联网接入选项、系统架构、数据持久层的选择、业务耦合、认证授权、前端、性能加速等，这么多内容浓缩在一章中，对笔者来说是非常大的挑战；因此，本章的编写和校对花费了笔者不少时间。尽管如此，读者仍可能会觉得许多内容还没有介绍通透。对于这一点，读者只能够通过阅读书中介绍的各类开源项目的源码来补足了。

笔者计划将本书中记录的技术通过开源项目 EPIC 来实现，以便让读者可以有个基本运行测试环境来开发物联网应用系统，并在 EPIC 基础上实现水平系统扩展和增值服务。

在开源的世界里，不是没有选择，而是选择太多。有了基本的思路和方向，选择开源组件构建自己的服务器就相对容易多了。

第 10 章

融合应用与数据分析

在已经介绍了物联网的系统构成、设备设计、服务器设计的基础上，本章内容主要涉及物联网系统如何与第三方应用进行数据交换和构成新的融合应用，以及相关的数据统计、数据分析、数据挖掘。在数据分析领域还出现了机器学习、人工智能的发展趋势，但这超出了本书范围，就不介绍了。

10.1 物联网是可编程的

Mashup，译成中文有“混搭”、“糅合”、“混聚”的意思，笔者还是选择“融合”一词来表达。Mashup 是 Web 2.0 时代出现的一种网络应用现象。最初 Mashup 一词主要用于音乐、文化、教育的融合，而 Web Mashup 特指融合两种以上的数据来源，在新的用户界面中形成一个全新的整合型应用。

物联网应用，在感应层进行数据采集，通过网络层将数据上传到应用层服务器和 APP，形成完整应用。除了设备采集的数据，数据还来自自有运维数据，并可以通过 Web API 整合第三方数据，或者通过网络爬虫收集各类数据来构建自己的大数据库，并在此基础上进行数据分析，最终通过 Web API 将服务提供给企业内部和外部用户或者第三方机构，形成一个完整的平台生态。这正是许多公司和机构努力的目标。

国外有一个网站的名称是 Programmable Web，这个名称暗示：Web 是可编程的。毋庸置疑，IoT/IoE 也应该是可编程的。随着越来越多的设备制造商构建自己的物联网云服务，其产生的大量物联网数据除了满足自己的业务要求之外，还会考虑将数据变现，或者交由第三方进行分析处理。

让我们观察可编程互联网的实例：最典型的例子就是谷歌地图。谷歌地图将切片后的地图通过 Java Script 嵌入任何网站中，实现了这些网站本身业务（如餐饮）与地理位置的融合应用。谷歌之后，大量公司投入 LBS 业务中，并拓展了其他形式的数据，包括街景、3D 建模和 VR。

10.1.1　Web API 的“满汉全席”

网站数据交换的手段主要是 Web API。国外的 Programmable Web、国内百度的 Web API 索引服务均提供了很好的参考。感兴趣的企业可以仔细观察，参考实施。

在 Programmable Web 上，地图一度是最大宗的 Web API 之一。主流 API 分类则是：

- 天气；
- 地图；
- 移动；
- 交通；
- 社交。

其中与 IoT 有关的 API 如下。

- IJENKO IoE² IoT API。用于构建 B2B、智能家居、能源管理应用；也可以构建消费应用，如控制智能加热、灯光和音乐自动化功能。REST API 可以用于数据分析或创建使用 IoT 的智能设备。
- MoBagel。提供了 IoT 实时数据分析的 API，并提供 Python、Java、Node.js、iOS 的 SDK。
- Mode API。为开发 IoT 应用环境而提供云平台。其采用 JSON 格式和 API Key 认证机制，功能包括多应用开发和访问控制。其由前 Google/Yahoo 员工推出，目标是成为“IoT 公司的虚拟后端”，并提供两套服务方案。
- SenseIoT API。SenseIoT 可以在云计算平台中安全、可靠地存储和处理传感器数据。它配备了数据可视化、设置触发器与远程管理数据的工具。SenseIoT API 使开发人员能够将任何设备、传感器或联网传感器网络与平台进行整合。
- GETHealth API。GetHealth 是从可穿戴及其他设备中收集健康数据的统一 REST API。它允许用户整合多个账户下不同的数据来源：Dailymile、Fatsecret、Fitbit、Google Fit、Jawbone、Lifefitness、MapMyFitness、MapMyRun、MapMyWalk、Microsoft Health、Misfit、Moves、PredicctBGL、Runkeeper、Strava、Sony Lifelog、Withings 及 23andMe。它还支持对于医疗设备的系统集成，并提供活动量、糖尿病、体重、营养、睡眠和基因的各种类型 API。此外，它解决了单一登录用户调用多个数据源时的用户身份验证问题。所以它可以跨越数据源，并提供统一的数据结构。

如果前几组 API 是传统的 RPC 调用或设备云应该具备的 REST API 的话，那么 GETHealth API 就是医疗健康领域的 Facebook 和 Twitter。

相比之下，国内的 API 聚合网站提供的 API 数量较少，质量有很大提升空间。

10.1.2 Web API 技术演进

Web API 是物联网实施中的关键技术之一，它不仅仅用于连接内部组件，还是服务器与外界的主要接口。目前，较为轻量的 JSON 替代 XML 成为 REST API 的主流序列化标准。因为大部分主流编程语言都支持 JSON 编解码，所以 JSON 技术对于数据供需两侧没有特别的技术要求。

除了 JSON，JavaScript 本身也可以实现类似的功能。地图类的 API 因为涉及版权的问题，没有采用 JSON 提供数据，而是提供 JavaScript API 给客户网站 JavaScript 使用，以避免己方图片信息被直接盗用。这里还需要实现 JavaScript 代码的跨域问题。需要先下载脚本，然后由脚本再加载跨域资源。这个领域基本上被 JavaScript 所主导，本书不深入讨论相关内容。

笔者在物联网设备云中采用同样的 API 方式，为客户应用云、移动 APP 和第三方平台提供数据服务。

10.1.3 IoT Web API 的必要性

面向物联网的 Web API 的需求和场景主要如下：

- 数据分享与融合，包括提供数据给第三方，以及从第三方获取数据；
- 数据分析，将原始数据提供给专门的分析公司；
- 数据可视化，将分析结果以图、表、3D 或动画等形式交付给用户。

许多数据单独分析意义不大。比如在医学领域，针对单台呼吸机设备进行数据采集仅仅是某个用户在某段时间内的使用数据，这就很难判断其实际健康状况。如果结合血氧、脉率的采集数据，则可以通过数据来了解呼吸机的使用与鼾症治疗疗效的关系。如果再结合血氧和血压监测，则可以了解鼾症、呼吸暂停、血压和血氧含量之间的联系。这样，数据采集和分析对于病患诊断和提高疗效才具备应用价值。如果再结合环境监测，如 PM2.5（细颗粒物）指数、新药临床双盲实验、流感疫情监控、天气预报、地理位置，那么设备的使用效果和实际意义才会凸显出来。要实现这一点，不仅需要物联网本身的原始数据采集，还要从其他行业中获取、引用、处理、推导和预测或规避某些事件。

在现阶段的实际工程中，笔者发现物联网原始数据的转发是个较大的需求。物联网设备大多都由供应商单独构建使用闭环：包括设备和服务器。以医疗设备为例，心脏监护仪表和血氧仪表分属两家供应商，他们均有各自的设备云服务器。但是医生需要整合数据进行诊断，这样势必会出现第三个服务器或者 APP，从两家供应商服务器获取数据。当大量原始数据在多个服务器之间传输时，不仅对现有服务器架构提出了挑战，还会造成带宽成本快速上升。

在这种压力下，会产生何种商业和技术变化？从原则上说，我们不应该移动数据，而应该移动算法。即使在边缘计算平台中，计算平台与设备之间、计算平台内部组件之间，也必须通

过 Web API 进行应用融合。

在数据可视化中，通过 Web API 提供 JSON 数据，可以很容易地实现基于浏览器的历史数据和实时数据绘图，以及其他形式的图表，如地图、统计分布图等。其实现基础也是 Web API 的数据分享。

综上所述，物联网具备 Web API 是必需的，物联网数据必须可编程。

10.1.4　Device as a Service

借用一下云计算领域的 XaaS 语法，就物联网来说，对于设备制造商将其设备数据脱敏后所提供的数据服务，我们可以称之为 DaaS：Device as a Service，设备即服务。在此实现的设备抽象化基础上实现设备可编程和更高层次的系统整合。

DaaS 如果成为现实，则意味着物联网设备，如门禁控制系统、电梯运行系统、广告推送系统、智能停车系统、支付系统等设备可以通过 Web API，将设备的 CRUD 权限和设备数据以免费或有偿的形式提供给第三方合理利用，并产生新的应用场景，如：

- 高速公路控制点的 LED 条屏可以通过获取附近智能停车系统、轨道交通系统的数据指导分流；
- 车联网提供的数据可以与餐饮行业、旅游行业相结合以产生资源优化配给；
- 将学生人群的手机位置数据作为家长微信群的通知数据；
- 体育场馆、高铁和地铁的手机位置数据可以作为交通企业的数据来源和运营依据；
- 空气净化器的 PM2.5（细颗粒物）传感器数据可以提供给公众做个参考；
- 商场门禁系统传感器的数据，如数量及开关次数等可以作为该商圈商业地产价值的指数；
- 快递员的手机位置数据可以作为其他商业用户的数据来源；
- 停车场的空车位数据可以提供给驾驶员做参考。

将物联网数据通过 Web API 交换给另一家企业或行业，会产生比广告业更加巨量的业务收入。但 DaaS 的关注重点首先在于**安全**和**隐私**，其次是流量和计费问题。

物联网的安全非常让人担心。9.8.1 节已经举了温度传感器触发喷淋系统的例子，这说明即便是只读操作（Read Only）的传感器，也是有隐忧的。对于那些具备写操作的权限：Create/Update/Delete，如果没有完备的鉴权和数据加密算法以及安全管理流程，还是先缓一步再实施。这也是笔者推荐 SSL/TLS 的原因。标准化算法可以解决安全问题，并降低服务器设计难度。

隐私是人权的基本组成部分，也会涉及安全性。比如，通过检测智能化水、电、煤气表的读数可以知道房屋中是否有人。如果这部分数据暴露在外部也是细思恐极。目前业界通过数据脱敏来解决隐私危机。所谓数据脱敏就是移除个人特征信息，具备匿名和群体统计特征的数据。

流量问题也是需要关注的，因为数据交付流出流量，所有的供应商都会对流出流量计费。控制数据流量要从设计角度解决。API 的设计原则如下：

- 有限的数据总量，限制数据库查询的范围；
- 有限的使用频率，监控 API 的使用频率；
- 有限的使用次数，监控 API 的使用总数；
- 紧凑的 API 设计，避免不必要的属性；
- 压缩传输，采用 gzip 等压缩方式传输数据，可实现 XML/JSON 文件的 30：1~10：1 压缩比。

在笔者的工程中，有客户要求 API 能够提供较为宽泛的查询条件，比如能够检索一个月的所有时序信号。对于此类要求我们在答复时需要非常谨慎。因为过大的数据集不仅会造成流量费用失控，还会形成慢 SQL，拖慢系统整体性能，甚至对外暴露系统弱点。为了防止滥用 API，必须设置限制。解决的方案是，将整月的数据切割为一天的数据，而且采用压缩方式传出。这种实施方式带来的好处如下：

- 慢 SQL 查询切割成多个普通 SQL 查询；
- 单个查询数据集有限；
- 流出流量被压缩。

计费问题需要依赖于 Web API 的使用监控，这是一个单独的子系统。初期计费策略可能会比较简单，但会随着市场竞争出现各种促销，导致计费算法复杂化。

物联网设备云通过 Web API 实现数据交换和应用融合是一种有商业价值的服务，但如何吸引第三方加入交换平台是一件需要资金和说服力的工作。此外，数据和服务的交付形式需要调研客户服务器和设备的具体需求而定。

10.2 数据统计、分析和挖掘

在编写本章内容时，笔者查阅了大量资料，发现了一个事实：**数据统计、分析、挖掘**领域的术语混用现象是令人咋舌的。这也说明一个事实：它是一门多领域交叉学科。但是无论如何，用户均希望从这些数据中得到后见之明、先见之明、洞察力和全局视角（hindsight，foresight，insight，overview），这一点是毋庸置疑的。

10.2.1 名词解释

统计学

统计学是研究数据收集、分析、诠释、呈现、组织方式的学科，可以在科学、工业或社会

多个领域应用统计学。统计学起始于人口统计或统计模型过程研究。该学科涉及统计处理数据的方方面面，包括数据采集规划，如调查表格的设计和实验数据等。

数据分析

数据分析是一个过程，覆盖数据检查、清洗、转换，以及根据数据利用目的，寻求建议结论与决策支持而采用的数据建模方法。数据分析可以得到事实的多面性，并可以使用多种分析手段实现。

数据挖掘

数据挖掘是一种特殊的数据分析技术，专注于数据建模及预测目的的知识发现。数据挖掘技术是计算机科学的交叉学科分支。它是在大型数据集中发现数据模式的计算过程，涉及人工智能、机器学习、统计和数据库系统的多种方法。数据挖掘过程的总体目标是从一组数据中提取信息，并把它变成可以理解的结构，以便进一步利用。除了原始数据的分析，还涉及数据库和数据管理方面，包括数据预处理、模型和推理、兴趣度度量、复杂性考虑，以及已发现的结构、可视化和后处理。

数据集成

数据集成是指把不同来源、格式、特点性质的数据在逻辑上或物理上有机地集中，从而为企业提供全面的数据共享。它是数据分析的前提。这里，数据分析也是数据建模的同义词。

10.2.2　术语小结

根据这些名词解释，我们可以得到如下结论：

- 所有这些与数据相关的流程、方法被统称为数据分析；
- 数据分析最初的数据格式化、预处理和接口整合，被称为数据清洗和数据集成；
- 数据“统计”是利用统计学方法进行的数据分析，它是数据分析中的一类；
- 数据“挖掘”是利用机器学习和人工智能算法进行的数据分析，它也是数据分析的一类。

以美国“9・11”恐怖袭击为数据分析案例，我们厘清这些概念：

- 通过数据（如所有人员的来源、资金和培训情况）采集和寻找对比，情报机构获得的“线索”都是后见之明（hindsight），因为事后看这些恐怖分子的关联线索很明显。
- 通过统计学方法，我们可以知道恐怖袭击分子的宗教、国籍、种族等信息，并可以利用统计学算法推算哪些群体是产生恐怖分子的高危群体。这是预见（foresight）能力。
- 通过数据挖掘，得到这些恐怖分子的社群网络、信用卡信息、电话交流、电邮、访问网址，我们可以推算出恐怖分子的行为模式和其他有用信息。这是所谓的洞察力（insight）。

- 通过数据可视化，我们可以了解恐怖分子的其他特性。比如关于经纬度信息，只有在可视化之后，我们才能知道恐怖分子的地理位置分布和主要来源；在现金流和社交网络数据可视化之后，我们才能推算出其现金流向和组织关系，这样我们就可以拥有全局视角（overview）。

这是一个数据分析的实例。由于现在数据源的丰富，因此在此基础上的“大数据分析”是个热门话题。现实生活中的许多产品也会尽量使用数据分析与统计来实现某些应用。手机 APP 中标注和提示“诈骗”和“骚扰”电话就是基于众包模式采集数据，并利用用户的标签数据进行数据统计来判断来电号码属性的。公安部门也可与电信部门和商业银行联手构建大数据分析平台，分析电信话务和银行账户的使用模式来预判、预防电信类诈骗活动。

10.2.3 大数据分析

“大数据”被越来越多的工程师、科学家、企业家所提及。许多企业意识到“数据即资产”，误以为自家企业从物联网中收集的数据就是大数据，其实并非如此。大数据有以下特点：

- Volume，体量“庞”大，往往在 TB/PB 级别；
- Variety，种类繁“杂”，数据源类型多，半结构化、准结构化、非结构化数据多；
- Velocity，速度飞“快”，尤其是增长速度快；
- Value/Veracity，真伪存“疑”，价值密度低，需要从巨量原始数据中“沙里淘金”，获取有价值的分析结果。

数据分析的概念并不新，大数据的重点在于传统分析方法手段在超大数据集面前已经失效，必须利用全新的云计算分布式集群、网络存储、并行分析算法和人工智能算法来实现数据分析目的。

随着大数据时代的来临，大数据的布道者很多，但是务实的教程却很少。原因很简单：能够深入讲解大数据分析的人，必须具备以下几个条件：

- 具备数学或统计学背景，懂得算法，了解 SPSS/SAS/R 等统计分析工具。
- 对于大数据在商业应用和相关技术方案方面有全局性的了解。
- 拥有真正的大数据来源。
- 拥有大数据的硬件平台，熟悉 Hadoop/Spark 等平台。学习环境和生产环境存在很大不同，有些问题只会在工程中暴露。
- 有清晰的大数据分析需求，了解所在的行业业务，需求明确后才可能开发算法。
- 在生产环境中使用大数据实务的一手经验。

同时具备这些条件的人，大多数来自大型企业。此外，大数据不仅仅应用于社会科学领域，其在医学和科学工程计算领域也有应用。比如，医学药品筛查、DNA 测序等都属于大数据分析领域。

10.3　采集整理自有数据

9.11.4 节提到的 EPIC 是笔者在物联网接入端设备云方面的一个尝试。将设备云服务器单独运行、独立于业务应用服务器，是可编程网站的具体实践，也几乎是行业标准做法。设备云主要解决的是设备数据采集、服务数据收集、简单数据统计，数据分析并非其重点。之所以将某些简单的统计和分析前置到设备云服务器中，主要原因是要满足小型系统的数据分析需求。

虽然笔者在大数据领域经验有限，但有关设备云服务器本身的数据采集、统计和分析的需求与实现，却可以与读者分享一下。

10.3.1　原始设备数据

在互联网应用中对用户行为的数据分析非常看重，因为这直接影响着一家互联网企业的业务变现能力。在物联网时代中，由于服务器面对的对象从“用户”变成了“设备+用户”，因此其分析方法主要围绕着设备和用户的使用和故障诊断而展开。

任何单一品种的联网设备，无论是工业门、呼吸机还是无人机，其收集的物理量都是比较有限的：

- 工业门——即时状态、开关频次、振动、受冲击报警、非法拆解、地理位置、电力供应情况等；
- 呼吸机——压力、流量、呼吸数据、使用时间和规律、地理位置等；
- 无人机——地理位置、姿态、负荷、电力供应、任务、报警、轨迹、任务、图传等。

读者可能觉得这里收集的数据也不算少了，但这与真正意义上的大数据相比，还是“小巫见大巫”。这些物理量数据属于结构化的时序数据，分析起来比较容易，主要使用各种统计学分析方法。

以呼吸机为代表的医疗设备为例，除了设备端采集上传的原始物理量需要记录或保存快照之外，还要即时计算各生化物理量的移动平均值以及脉冲波形的脉宽、间隔、斜率等。如果这些根据衍生计算数据触及预设的触发条件，则还会对触发事件进行记录和转发推送。如果需要更多可定制的算法，前端数据收集应该将数据直通到实时大数据平台中进行处理和记录。

10.3.2　数据埋点

所谓“埋点”，指在正常的功能逻辑中添加统计逻辑。这些埋点往往位于一些关键的节点之上。用户在使用 HTML5 网页、APP 或硬件时，通过这些埋点可以收集到用户的使用行为模式以及流量转化率等信息。通过数据埋点采集到数据之后，我们就可以进行批量处理、日志记录、数据入库，以及进行统计分析与可视化。物联网往往依赖于硬件设备，不同于 Web/APP；但在

某些复杂设备中，依然可以制作一些“埋点”。对于按键、开关以及其他物理量进行收集，可以了解设备的使用效果。

10.3.3 服务器端数据

除了采集远程设备上传的原始数据，还需要采集服务器端即时或周期性使用数据。这些数据与传统互联网应用的服务器运维数据是类似的。

业务数据

具体数据如下。

- 设备全局数据：总数、激活数量、活动数量、最大数据数量、IP 统计数量、语种统计；
- 设备个体数据：连接时间、断开时间、累计传输报文；
- 用户全局数据：总数、激活数量、活动数量、最大在线数量、IP 统计数量、语种统计；
- 用户个体数据：登录时间、登出时间、IP 地址、活动；
- API 全局数据：总数、分类排名；
- API 个体数据：调用次数、IP 地址、推荐地址；
- 日志数据：错误、报警、外部攻击、电邮统计。

这里的数据有部分是即时收集的，比如连接和断开事件发生之际就进行记录；有些需要周期性统计，如连接设备数量等。后者虽然可以直接获得连线数量，但需要周期性收集连接数量的数据才能够了解连接数量的变化趋势。

运维数据

运维数据主要与服务器性能监控有关：包括 CPU/MEM 占比、磁盘占比、入网流量、出网流量、端口流量、系统进程、磁盘 I/O 和 SQL I/O 特性。这些信息主要依赖系统工具包和服务器日志管理系统实现数据收集和统计。

10.3.3.1 日志分析

在 Linux 平台中，有许多优秀的日志管理工具：LogStash、syslog 都是。不过不同的日志文件格式不同，所以需要脚本语言来定制。Python 脚本解析日志的三个步骤：读取、解析、入库。

1．读取日志

在笔者设计的 Twisted/Cyclone 组合中，日志都是 Twisted 格式的，相对统一。每个日志文件为 1MB。相对来说设备端流入的数据多，日志文件积累得较快。笔者单独设置了一个文件夹，每隔 20 分钟将所有日志文件移入这个文件夹做批量分析处理，处理完毕后直接删除。在文件读取中可以采用一行行读取的方式，也可以采用批量读取数据的方式。具体选用哪种方式和服务器负载、存储负荷有关联。

readline.py：

```
file = open("twistd.log.1".'r')
for line in file:
    print line
file.close()
```

readlines.py:

```
file = open('twistd.log.1','r')
lines = file.readlines(1000) # read 1000 lines
print lines
file.close()
```

一行行读取速度慢，但是节省内存；批量读取速度快，但消耗内存。

2．解析日志

由于现在许多服务都是以持续更新的方式存在的，因此日志中存在大量的调试信息。所以调试后最好将所有信息都关闭，并在解析日志的时候将其与正常日志区分开来。

```
2016-06-27 08:44:20+0800 [http] 200 GET /json/v1/devices (180.175.138.44) 1.44ms
2016-06-27 08:44:25+0800 [http] 200 GET /json/v1/devices (180.175.138.44) 1.53ms
2016-06-27 08:44:26+0800 [http] 200 GET /json/v1/usage2/A1H470029/2016-06-26 (117.
136.35.18) 11.97ms
```

在样本中，[http]是一个很好的分类标签。这倒过来启发笔者将其余调试目的的日志进行标签化。在设备端口采用新的分类标签：用[debug]、[attack]、[discon]、[conn]、[packet]、[dbg]区分不同的日志目的。这样，日志解析器可以很容易地判断并过滤掉不需要的日志。解析日志时可以采用正则表达式和 split 方法进行解析。在这方面 Python 很强，但是正则表达式需要根据需求进行设计。

```
>>>> line = "2016-06-27 08:44:26+0800 [http] 200 GET /json/v1/usage2/A1H470029/2016-
06-26 (117.136.35.18) 11.97ms"
>>>> print line.split(" ")
['2016-06-27', '08:44:26+0800', '[http]', '200', 'GET', '/json/v1/usage2/A1H470029/
2016-06-26', '(117.136.35.18)', '11.97ms']
```

也可以采用最简单的 grep 命令来过滤标签，并存入新的文件。

```
grep -rin "http" ./twistd.log.*
```

3．日志入库

日志入库，指的是录入数据库，方便日后统计页面的数据可视化。数据从日志记录到 SQL 数据库中是一个批处理的过程，所以要尽可能地做批量插入处理，而不要一条条数据插入。但是我们可以适当分割任务，以避免数据过载和慢 SQL。可以使用 executemany 方法，也可以采用自行拼接 SQL 字符串方式插入数据库。

```
import MySQLdb
import random
conn = MySQLdb.connect(host = '127.0.0.1',
       user = 'root',
       passwd = '123456',
       db = 'visit')
print conn
cur = conn.cursor()
li = list()
for i in range(1, 10000):
       idm = random.randint(1000, 10000)
       name = 'demo' + str(idm)
       li.append((idm, name))

sql = """insert into mytable values(%s, %s)"""
cur.executemany(sql, li)
conn.commit()

cur.execute("select * from mytable")
ret = cur.fetchall()
for x in ret:
       print x
cur.close()
conn.close()
```

不仅仅是 MySQL/Maria 之类的 SQL 数据库，Redis 也支持批量处理。Redis 数据库插入数据操作是非常快的，批量处理的目的是减少建立 TCP 连接所导致的系统消耗。

如果日志系统非常复杂，则可以利用多个工作线程：读取、解析、入库各使用一个线程，三者之间采用消息队列进行异步通信，实现整个流程的流水线操作。

10.3.3.2 实现方式

许多数据，包括设备数据、服务器业务数据和运维数据是实时更新的。比如，即时连接数量、最大连接数、CPU 占比等，可以作为一个全局变量保存在服务器中。但这些数据需要周期性地采集、分析并记录在数据库中。我们可以将这些数据采集线程都合并在原有设备云服务器中。但随着系统需要收集的数据越来越多，整个系统会变得难以维护。所以我们更加倾向于设计一个松耦合系统，由单独线程定时通过 Web API 去获取这些信息。

正确的做法是利用消息队列组件来设计分布式多进程任务调度系统。即：

- 接入系统，也就是业务系统负责设备连接；
- 将时间触发任务设计为独立程序，即周期任务，例如定时收集服务器和运维信息、数据归档和日志统计；
- 将事件触发任务设计为独立程序，即异步任务，例如电邮和短信报警；

- 使用 crontab/Plan 或者 Celery beat 调度周期任务；
- 使用消息队列或者 Celery/RQ 等组件耦合异步任务、周期任务和业务系统；
- 周期任务、异步任务的调度结果可以通过消息队列推送回调用者，并保存在数据库中；
- 用户通过 Web GUI 进行任务调度及查询任务处理结果。

松耦合系统可以单独升级其中一部分设计，而不会影响到其他系统。而且随着系统的复杂化，许多进程都单独运行在单独的服务器或者虚拟的云服务器集群之上。客户端和服务器端只需要满足既定的协议，就可以各自升级，而不会影响对方的运行。

10.3.4　需求确定分析方法

在实际生产活动中，企业最初的数据需求往往是不明确的。常常有企业主对笔者说：“您能不能替我们构想一下大数据分析能够实现什么？” 在企业所在的领域，企业方本身是业务方面的专家，却问一个对该领域不熟悉的人，这是本末倒置了。正确的方向是，由企业方表达大数据的使用愿景，由开发者来实现它。

在物联网应用中，尤其是在 B2B 的应用中，其物联网应用的核心诉求是，流程优化、成本优化、产品改进，以促进销售为目的。但是落实到具体数据集和分析方法方面，则需要企业方和数据分析师坐下来探讨。

现实情况是，数据分析的目的不明确，数据收集后大量数据无法处理，每日占用了大量存储资源，而无法变成对业务有用的数据，这造成了一种持续增长的支出。所以在小系统中，笔者一般推荐采取保留部分数据快照的形式，以保留最新的数据子集。必须在确定了数据分析需求后，才可以定制对应的数据分析算法和可视化方法，并持久保存各类数据。

即使在小规模使用范围内，数据分析也需要在物联网设备云服务器的数据统计和业务应用（ERP/CRM）服务器整合之后才能够进行数据分析。比如超市的数据分析往往需要先基于传感器和 RFID 系统收集、统计顾客的行为模式，并整合库存物流系统数据，通过数据分析得出当前阶段的畅销商品和滞销商品，这样才能够有针对性地刷新电子货架标牌，进行促销推广等。

所以，需要先确定分析需求，之后在收集的原始数据基础上可以进行各类数据分析，并集成其他的自有数据来源，进行数据挖掘。

10.4　采集第三方数据

除了努力整合自有数据来源，获取第三方数据也是很重要的事情。我们需要仔细规划，而且需要时刻应变。即使是购买了第三方的数据服务，也可能由于数据供应商的原因而中断服务，因此无法继续支持自己的客户而导致业务损失。Python 及 JavaScript 爬虫可以帮助企业和个人

获取更多的信息资源，这是一种我们必须掌握的技能。网络爬虫和运维一样是 Python 的传统优势应用领域。

网络爬虫用于一个斗智斗勇的场合。网站之所以不愿意通过 Web API 提供内容和数据，原因之一就是不愿意分享数据。所以当有外部爬虫抓取数据时，一方面会产生大量的数据出网流量，另一方面网站方的内容和数据也容易被完整复制。所以，网站方会尽可能地采用各种手段来防止爬虫抓取内容和数据。但网站本身也需要各类搜索引擎来抓取数据以实现推广的目的。所以网站对于网络爬虫和搜索引擎的心态是非常矛盾的。为此，互联网行业确立了 robots.txt 格式，通知网络爬虫哪些路径是可以抓取的、哪些路径是禁止访问的。网络爬虫是一种技术手段，开发者需要遵守目标网站的使用条款、robots.txt 约定和相关法律法规。

10.4.1 结构化数据

结构化数据，即存储在数据库中，可以用二维表结构来表达的数据。一般而言，结构化数据大部分存储在 SQL 数据库中，属于网站的核心数据。早期网站多以 HTML 表格提供内容，并可以遍历整张表格。如果爬虫能够遍历数据库，则意味着其已经攫取了网站的核心价值。现在大多数网站一方面多以 AJAX 方式提供内容，另一方面对于数据库遍历做了许多限制。爬虫设计者需要预先仔细了解目标网站的 robots.txt，以免侵犯目标网站的知识产权。Excel 和 CSV 也是二维表格，可以归类于结构化数据。这是爬虫可以抓取的内容。

10.4.2 半结构化数据

半结构化数据往往结构和内容混在一起，没有明显区别。但是其可以较为容易地转化为结构化数据。

HTML、XML 属于半结构化数据，而 XHTML 是 XML 修饰过的 HTML 文件。XML 来自 SGML，其好处是将内容与实现脱耦，而且容易与其他应用进行数据交换。其缺点是作为标记语言，文件尺寸有些大。这类 XML 文件虽然归于非结构化数据，但是却很容易使用 DOM 工具转化为结构化数据。这也是早期 Web Service 利用 XML 作为接口语言的原因之一。

除了 XHTML，XML 的其他著名格式如下：

- RSS/ATOM，新闻聚合内容格式；
- GeoRSS/KML，地理位置文件；
- SVG，可缩放矢量图形文件；
- SMIL，同步多媒体语言。

由于 XML 的半结构化特点，因此其携带的数据具备可交换性和通用性。除了 XML 文件外，JSON 格式还可以实现 XML 的大部分功能。此类接口或文件是爬虫的重要获取对象。

10.4.3　非结构化数据

不方便用数据库二维逻辑表来表现的数据即被称为非结构化数据，包括所有格式的办公文档、文本、图片、各类报表、图像和音频/视频信息等。

非结构化数据库是指其字段长度不等，并且每个字段的记录又可以由可重复或不可重复的子字段构成的数据库。用它不仅可以处理结构化数据，如数字、符号等信息；更可以处理非结构化数据，如全文文本、图像、声音、影视、超媒体等信息。

非结构化 Web 数据库主要是针对非结构化数据而产生的。与以往流行的关系数据库相比，其最大区别在于它突破了关系数据库结构定义不易改变和数据定长的限制，支持重复字段、子字段以及变长字段，并实现了对变长数据和重复字段进行处理与数据项的变长存储管理，其在处理连续信息（包括全文信息）和非结构化信息（包括各种多媒体信息）中有着传统关系数据库所无法比拟的优势。

利用网络爬虫抓取这些数据的策略是不同的：

- 结构化数据——分析 URL、遍历表格、解析表格内容、数据入库；
- 半结构化数据——分析 URL、下载数据、解析数据、数据入库；
- 非结构化数据——分析 URL、下载数据、保存文件。

10.4.3.1　Scrapy 爬虫

Python Scrapy 是一种基于 Twisted 框架的网络爬虫。传统的 HTML 网页，即使是 XHTML，由于混杂了各种内容，也应该归类于非格式化数据和半结构化数据。配合 BeautifulSoap 等额外的处理包，爬虫可以处理抓取到的 HTML 网页，并从中提取有用的信息。Scarpy 很适合抓取那些设计比较原始的网站。其典型特征如下：

- 采用 HTML 表格方式提供数据库中的数据；
- URL 路径采用简单 ID，容易遍历数据库。

随着越来越多的网站采用 JavaScript/AJAX 等技术升级其 Web，甚至采用单页 JavaScript 的设计，简单 HTML 表格网页设计在逐年减少。一些设计会针对网络爬虫设计一些逻辑陷阱，让爬虫在同一网页中“兜圈子”。

1．安装

安装命令如下：

```
pip install scrapy
```

2．依赖项目

Scrapy 依赖于 Twisted、PyOpenSSL、queuelib、lxml，可能需要源码编译。所以需要在 Windows 下事先安装 Visual C++，或者从以下网址下载预编译版本：

- pywin32：http://sourceforge.net/projects/pywin32/files/；
- Twisted：http://twistedmatrix.com/trac/wiki/Downloads；
- zope.interface：采用 pip 安装就足够了；
- lxml：http://pypi.python.org/pypi/lxml/；
- pyOpenSSL：https://launchpad.net/pyopenssl。

3．创建 Scrapy 项目

运行以下指令：

```
$ scrapy startproject democrawl
$ cd democrawl
$ scarpy genspider example example.com
```

产生新的爬虫项目目录，如图 10-1 所示。

```
democrawl/
├─democrawl
│  ├─__init__.py
│  ├─__init__.pyc
│  ├─items.py
│  ├─pipelines.py
│  ├─settings.py
│  ├─settings.pyc
│  └─spiders
│     ├─example.py
│     ├─__init__.py
│     └─init__.pyc
└─scrapy.cfg
```

图 10-1　项目目录

- scrapy.cfg：项目的配置文件。
- democrawl/：项目的 Python 模块，可在该文件夹中添加代码。
- democrawl/items.py：项目的 items 文件。
- democrawl/pipelines.py：项目的管道文件。
- democrawl/settings.py：项目的配置文件。
- democrawl/spiders/：spider 所在的文件夹。

4．定义采集条目（Item）

我们的检索目标是 dmoz 网站目录。

democrawl/items.py：

```
import scrapy
```

```
class DemocrawlItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    title = scrapy.Field()
    link = scrapy.Field()
    desc = scrapy.Field()
```

5．编写爬虫

将 example 更名为 dmoz_spider.py。

democrawl/spider/dmoz_spider.py：

```
# -*- coding: utf-8 -*-
import scrapy


class DmozSpider(scrapy.Spider):
    name = "dmoz"
    allowed_domains = ["dmoz.org"]
    start_urls = (
        'http://www.dmoz.org/Computers/Programming/Languages/Python/Books/',
        'http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/'
    )

    def parse(self, response):
        filename = response.url.split("/")[-2]
        open(filename, 'wb').write(response.body)
```

6．运行爬虫

```
$ cd democrawl
$ pwd
/root/python_book/source/scrapy/democrawl
$ scrapy crawl dmoz
```

请注意路径深度，若路径不对，则会提示 Scrapy 没有 crawl 指令。此时爬虫启动，终端中会显示以下数据：

```
2016-06-29 17:14:27 [scrapy] INFO: Spider opened
2016-06-29 17:14:27 [scrapy] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items
 (at 0 items/min)
2016-06-29 17:14:27 [scrapy] DEBUG: Telnet console listening on 127.0.0.1:6023
2016-06-29 17:14:28 [scrapy] DEBUG: Crawled (200) <GET http://www.dmoz.org/robots.tx
t> (referer: None)
2016-06-29 17:14:29 [scrapy] DEBUG: Crawled (200) <GET http://www.dmoz.org/Computers
/Programming/Languages/Python/Resources/> (referer: None)
2016-06-29 17:14:29 [scrapy] DEBUG: Crawled (200) <GET http://www.dmoz.org/Computers
/Programming/Languages/Python/Books/> (referer: None)
2016-06-29 17:14:29 [scrapy] INFO: Closing spider (finished)
```

```
2016-06-29 17:14:29 [scrapy] INFO: Dumping Scrapy stats:
{'downloader/request_bytes': 734,
 'downloader/request_count': 3,
 'downloader/request_method_count/GET': 3,
 'downloader/response_bytes': 16830,
 'downloader/response_count': 3,
 'downloader/response_status_count/200': 3,
 'finish_reason': 'finished',
 'finish_time': datetime.datetime(2016, 6, 29, 9, 14, 29, 188009),
 'log_count/DEBUG': 4,
 'log_count/INFO': 7,
 'response_received_count': 3,
 'scheduler/dequeued': 2,
 'scheduler/dequeued/memory': 2,
 'scheduler/enqueued': 2,
 'scheduler/enqueued/memory': 2,
 'start_time': datetime.datetime(2016, 6, 29, 9, 14, 27, 359800)}
2016-06-29 17:14:29 [scrapy] INFO: Spider closed (finished)
```

此时路径中多了两个文件：Books.html/Resources.html。

7．访问网页中的特定内容

下载 HTML 文件不是我们的目的，我们需要采用 HTML 开发工具查看 HTML 网页内容，并利用 CSS 选择器或者 XPath 选择器语法来访问所需目标字段。为了更好地使用 XPath，Scrapy 提供了一个 XPathSelector 类，它有两种方式：HtmlXPathSelector（HTML 相关数据）和 XmlXPathSelector（XML 相关数据）。选择器有以下三种方法。

- select：返回选择器的列表，每一个 select 表示一个 XPath 表达式选择的节点。
- extract：返回一个 unicode 字符串，该字符串是 XPath 选择器返回的数据。
- re：返回 unicode 字符串列表，字符串作为参数由正则表达式提取出来。

除了网页开发工具，开发者可以采用 Scrapy shell 来查看 XPath。

```
scrapy shell http://www.dmoz.org/Computers/Programming/Languages/Python/Books/

[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0xa837dcc>
[s]   item       {}
[s]   request    <GET http://www.dmoz.org/Computers/Programming/Languages/Python/Boo
ks/>
[s]   response   <200 http://www.dmoz.org/Computers/Programming/Languages/Python/Boo
ks/>
[s]   settings   <scrapy.settings.Settings object at 0xa83748c>
[s]   spider     <DmozSpider 'dmoz' at 0xab8fecc>
[s] Useful shortcuts:
[s]   shelp()          Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local objects
```

```
[s]   view(response)    View response in a browser
>>> response.header
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'HtmlResponse' object has no attribute 'header'
>>> response.headers
{'Cteonnt-Length': ['52474'], 'Content-Language': ['en'], 'Set-Cookie': ['JSESSIONID
=13228DB7ADAA1DDA2D7A7AB5070B9FE6; Path=/; HttpOnly'], 'Server': ['Apache'], 'Date':
 ['Wed, 29 Jun 2016 09:31:39 GMT'], 'Content-Type': ['text/html;charset=UTF-8']}
>>> response.xpath("//title")
[<Selector xpath='//title' data=u'<title>DMOZ - Computers: Programming: La'>]
>>> response.xpath("//title").extract()
[u'<title>DMOZ - Computers: Programming: Languages: Python: Books</title>']
>>> response.xpath("//title/text()").extract()
[u'DMOZ - Computers: Programming: Languages: Python: Books']
>>> response.xpath("//title/text()").re('(\w+)')
[u'DMOZ', u'Computers', u'Programming', u'Languages', u'Python', u'Books']
```

若欲了解 XPath 语法详情，则可查看本章延伸阅读部分中的教程。同时将更新我们的 dmoz_spider.py：

```
# -*- coding: utf-8 -*-
import scrapy


class DmozSpider(scrapy.Spider):
    name = "dmoz"
    allowed_domains = ["dmoz.org"]
    start_urls = (
        'http://www.dmoz.org/Computers/Programming/Languages/Python/Books/',
        'http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/'
    )

    def parse(self, response):
        for sel in response.xpath('//ul/li'):
            title = sel.xpath('a/text()').extract()
            link = sel.xpath('a/@href').extract()
            desc = sel.xpath('text()').extract()
            print title, link, desc
```

如果读者的实验是失败的，那么网络爬虫的对象网站一定已经升级，或者至少改动了网页的 DOM 结构。

8．保存爬虫内容

Scrapy 可以使用-o 选项导出 JSON 格式的数据。如果 Scrapy 整合 SQL 或 NoSQL 数据库，则通过 Item Pipeline 可以实现更多功能。需要注意的是：Scrapy 基于 Twisted，所以其数据持久

后端需要采用异步驱动，这一点可以参考 Twisted/Cyclone 的数据库驱动。

```
$ scrapy crawl dmoz -o items.json
```

Scrapy 爬虫是一个迭代很快的 Python 扩展包。笔者记得在 Scrapy 早期项目中，只能够利用 BeautifulSoap 来从 HTML 网页中提取字段，而现在已经发展到采用 XPath、CSS 以及 JavaScript 脚本插件来提取信息字段。除了在本地运行爬虫，Scrapy 团队还代为托管云中的爬虫以及进行商业服务。

10.4.3.2 支持 JavaScript 的爬虫

大多数网站或多或少都依赖于 JavaScript 及 jQuery 等 JS 库构建 AJAX 应用。这种设计往往是最简单的 XHTML 架构，并使用 JavaScript 进行数据填充。单纯的 XHTML 爬虫遇到这种网站就会比较“抓瞎”，因为其只能够看到 XHTML 框架，而无法抓取框架中的内容。所以，需要内置 JavaScript 的爬虫来应对此类网站。此类爬虫通用性强，但是需要加载 DOM 模型，并运行 JavaScript 脚本；其相对复杂，而且占用计算机资源。

常见的 JavaScript/Python 混合型架构包括 ScrapyJS、PyV8、PythonWebKit、Selenium、Splinter 等。纯粹 JavaScript 的爬虫有 Neocrawler。限于篇幅，请读者自行通过本章延伸阅读部分了解详情。

10.4.4 数据录入

不同数据的录入方式与数据本身有很大关联。无论数据来源是网络爬虫还是组织机构内部或者外部，都需要分别处理。结构化数据可以直接转换成现有数据库格式后入库，或者针对非结构化数据进行特征提取、建立索引、录入数据库。其主要手段如下：

- 提取声音指纹；
- 提取图像指纹；
- 提取地理位置；
- 提取自然文本中的关键字；
- 提取数据标签。

同时采用不同的数据库技术保存这些原始数据。

10.4.5 数据融合

在大数据时代，数据源是多样的、自然形成的；而海量的数据往往是半结构或无结构的。这就要求数据科学家和分析师驾驭多样、多源的数据，将它们梳理后进行挖掘和分析。在这个过程中，数据融合（data blending）就成为不可或缺的一步。对于不同格式的数据源，我们需要

采用不同的手段和方法。

美国国防部实验室对数据融合的定义是，将多传感器信息源的数据信息加以联合、相关及组合，获得更为精准的位置估计和身份估计，从而对战场态势和威胁及重要程度进行实时、完整评价的处理过程。

其新定义如下：以产生决策智能为目标，将多种数据源中的相关数据进行提取、融合、梳理整合成一个新的分析数据集（Analytic Dataset）。这个分析数据集是独立的和灵活的实体，可随数据源的变化重组、调整和更新。

数据融合有六个基本步骤：

（1）连接所需多源数据库并获取相关数据；

（2）研究和理解所获得的数据；

（3）梳理和清理数据；

（4）数据转换和建立结构；

（5）数据组合；

（6）建立分析数据集。

据统计，数据科学家 80%的精力会花费在前期的数据准备阶段。数据的来源可以归纳为以下三大类。

- 主要数据（Primary Data）：包括企业或组织直接采集、掌控的内部运行数据和营销数据。
- 次要数据（Secondary Data）：第三者采集、整理和提供的二手数据，如经济指标、人口普查、民意调查、网络数据等。
- 科学数据（Scientific Data）：包括科学研究的成果、指数、算法、模型等。

如何融合这些数据，非常考验数据工程师的建模能力。

数据融合有个典型的例子，大家可以参考：传感器数据融合。在无人机的飞行控制单元中，9DOF IMU/AHRS（惯性测量单元/航行姿态参考系统）需要将加速度计、陀螺仪、磁场仪数据进行数据融合，形成对于飞行控制有意义的数据。没有数据融合，传感器发出的数据将毫无意义。这方面有大量的文献可以查询，传感器多数据源融合采用的技术有卡尔曼滤波等，并大量使用浮点计算。

由于需要熟悉传感器特性并具备数据建模的知识，因此设计一款优秀的飞控软件的门槛比较高。Freescale 曾经单独出售过一款 IMU 控制板模块，板载多种传感器，并由板载 Cortex-M4 负责传感器数据融合输出，以简化用户的开发难度。一般来说，在开源活动这么普及的情况下，闭源并作为套件出售，说明该公司对于知识产权采用了保密的方式，这也是通过套件形式来简化和降低用户的使用门槛。所以，这件事情本身门槛比较高。不过现在看来，优秀的飞控设计的确不多。

10.4.6 数据规整

数据规整就是对原始数据进行加载、清理、转换和合并。通过数据规整，可以实现不同的数据在语义和格式上的一致性。

比如：软件的全面升级，肯定会带来数据库的全面升级，每一个软件背后的数据库的构架与数据的存储形式可能是不相同的，这样就需要数据的转换了。此外，由于数据量的增加或不合理的数据存储架构，无法满足数据检索和规模扩展需求，也需要数据转换。

数据规整可以使用通用编程语言，如 Java 和 Python；也可以使用在文本操纵方面有特长的语言：Perl/R；还可以使用 UNIX 下的文本工具，如 sed 和 awk。Python 标准库以及 pandas 等数据统计软件包覆盖了大部分的数据规整需求。

其主要手段如下：

- 合并数据集；
- 重塑和轴向旋转；
- 数据格式和精度转换；
- 字符串操作。

10.4.7 数据交易

数据融合必然伴随着数据交易市场。如何切入和构建平台是一件需要仔细规划的事情。2015 年 9 月 5 日，国务院印发了《促进大数据发展行动纲要》，系统部署大数据发展工作，这意味着数据作为基础性战略资源被纳入了国家重点规划。在此之前，已经有地方政府开始推动大数据产业的发展。2015 年 4 月，国内首个大数据交易所在贵阳挂牌成立，同年 7 月，武汉成立了长江大数据交易所，摸索如何将海量的数据变成“大数据”，建立完善的大数据交易规则。北京中关村数海大数据交易平台，是由中关村大数据交易产业联盟负责承建的数据交易服务平台，其是通过开放的 API 数据录入、检索、调用，为政府机构、科研单位、企业乃至个人提供数据交易和使用的场所。该平台已接入京东、新浪、天翼等开放 API 数据千余条，收录国外 API 数据达数千项。2016 年 2 月底启动以来，其已产生 85 笔交易，交易额达 112 万元。

通常撮合一个交易短则数月，长则一年。其中涉及的问题很多，包括需求端的要求是否合理、供应端的数据质量是否达到要求、IT 系统是否具有开放能力等。这些条件达成后，还有商务谈判、系统对接等问题，一个流程走下来，最快也要一两个月。所以，各个大数据交易所显示市场活跃度还有待激活。其中最大的制约因素是大数据交易的规则滞后，大数据产业落地的进程还在与各企业建立合作关系的阶段。总的来说，数据交易目前还没有形成规模效益。

10.5　数据分析

前面介绍名词解释以及数据采集、网络爬虫、数据规整等内容只为一个目的：数据分析。

数据分析是指用适当的统计分析方法对收集来的大量数据进行分析，提取有用信息和形成结论而对数据加以详细研究和概括总结的过程。这一过程也是质量管理体系的支持过程。在使用中，数据分析可帮助人们做出判断，以便采取适当行动。在实际应用中，我们看到数据分析在科学计算、工程分析、社会科学统计、金融分析、大数据分析领域得到了大量的应用。但是在这几种应用中分析需求和使用的分析方法、手段和工具集是完全不同的。

10.5.1　常见编程语言

在各类数据分析领域，常见的编程语言如下：

- R，作为免费软件，R 是 SAS/MATLAB 的替代品，但其在处理大数据时很费力；
- Python，工具集丰富，但处理大数据不够高效，需要 C++补充；
- Julia，非常年轻的语言，功能不够丰富；
- Java，大数据的基础语言；
- Scala，基于 Java VM 的高阶算法语言；
- MATLAB，著名的商用科学计算软件包，包括机器学习、信号处理和图像识别；
- Octave，MATLAB 的免费替代品；
- Go，“更好的 C 语言”，来自 Google。

在金融分析领域，R/SAS/MATLAB/Java/Python/C++是主要的编程平台。量化交易依然依赖于性能与速度，C++依然地位牢固。

可以得出一个结论：Python 在数据分析领域所占的份额远超其在嵌入式、桌面和服务器领域的份额。

10.5.2　数据分析分类

以应用目的来分类：

- 科学计算——针对特定场景中的某种物理量进行分析，需要转换到其他域中进行分析以得到工程目的结果，比如频谱分析、噪声分析、光谱分析等。
- 金融分析——针对交易时序信号进行分析，并配合其他随机冲击的变量进行分析。其目的是获得更高或者更确定的收益，并寻找其中的规律和交易手段，往往使用时域分析或者小波变换。
- 大数据分析——针对网络或物联网传感器收集的大容量数据进行分析，从而对产品、

人群行为模式等进行较为宏观的数据统计和分析，这里面统计往往是重头戏。

- 某些科学活动也需要大数据分析，比如药品分类和 DNA 分析等。

这种列举的方法很粗略，无法涵盖所有的数据分析领域。笔者列举这些数据分析种类，想说明一个事实：不同的应用，针对不同的目标数据集，存在不同的分析目的，这决定了其采用的分析方法、手段甚至平台存在很大差异。

10.5.2.1 科学计算数据分析

其中，科学工程领域的数据分析，必须要提到一家位于美国得州奥斯汀的软件公司：Enthought，该公司主要使用 Python 从事科学计算工具的开发。Enthought Tool Suite（ETS）包括 NumPy、SciPy、Traits、TraitsUI、Chaco、TVTK 以及 Mayavi 均为 Enthought 公司开发或维护的开源程序库。ETS 大大简化了与科学工程计算相关的计算、可视化、分析应用的开发难度，缩短了开发周期。除上述软件包外，Enthought 还有以下开源项目：

- Enaml（Enamal is Not A Markup Language 的缩写），用于创建专业质量 UI 的软件包。
- Enstaller，egg-based Python 发布管理和安装工具。
- Envisage，Python 应用框架，用于构建支持插件的可扩展应用。
- BlockCanvas，函数和数据分离的拖曳式仿真实验环境。
- GraphCanvas，复杂绘图的可视化与互动软件包。
- Enable，支持容器和事件通知的对象绘制库和 PDF 矢量绘图引擎。
- Casuarius，Cassowary 约束求解 Cython 绑定。
- AppTools，常见工具集合，封装了一些常见的 Python 标准库和扩展库。
- Encore，ETS 的核心工具模块，包括事件、存储、并发、终端等。
- CodeTools，主要包括元类编程，帮助开发者分离数据和代码。
- ETSDevTools，开发工具、调试、测试、检查代码工具。
- SciMath，支持科学和数学计算的软件包。
- ETSProjectTools，与 Subversion 版本管理有关的 Python 包。

Enthought 是一家值得我们尊重的机构，它几乎将自己所有的成果和内部实现细节均开源给大家使用了。其官方网址如下：

http://code.enthought.com/projects/index.php

10.5.2.2 统计学数据分析

统计学的基础是数学、概率等，包括：

- 描述性统计，包括均值、中位数、总数、方差、标准差和统计图表；
- 概率基本概念；
- 条件概率和贝叶斯公式；

- 微积分、随机变量及其分布，如二项分布、均匀分布、正态分布；
- 多维随机变量及分布；
- 方差与协方差；
- 大数定律、中心极限定理与抽样分布；
- 参数估计；
- 基于正态分布的假设检验；
- 秩和检验；
- 回归分析；
- 方差分析；
- 时间序列分析；
- 随机过程与马尔科夫链。

在统计学领域，有些人将数据分析划分为描述性统计分析、探索性数据分析以及验证性数据分析。其中，探索性数据分析侧重于在数据之中发现新的特征，而验证性数据分析则侧重于已有假设的证实或证伪。

探索性数据分析是指为了形成值得假设的检验而对数据进行分析的一种方法，它是对传统统计学假设检验手段的补充。该方法由美国著名统计学家约翰·图基（John Tukey）命名。

定性数据分析又被称为“定性资料分析”、“定性研究”或者“质性研究资料分析”，是指对诸如词语、照片、观察结果之类的非数值型数据（或资料）的分析。

读者可以从互联网上查阅到大量的统计学和大数据分析的书单。但是此类专业书籍大多基于统计数学，书中主要围绕基本概念和数学公式展开。对于数据统计分析完全无基础的读者，可以阅读一本比较容易阅读的书籍《漫画统计学入门》（辽宁教育出版社）。该书以漫画的形式介绍了统计基础概念。不过由于翻译的原因，书中的某些术语不太符合国情。但它依然是值得推荐作为读者快速了解统计学背景知识的入门书。

10.5.2.3　大数据分析

大数据分析的应用场景如下：

- 充分理解客户，满足客户需求，如了解购物习惯、预测购物周期、库存管理、保险公司产品精算；
- 业务流程优化，如根据话务流量、设备使用频率、运维周期等数据分析优化配置所需的人力资源、物流运能、工厂生产电力成本；
- 个人生活质量的改善，如热量和睡眠模式追踪、相亲匹配度；
- 医疗保健和研发，如统计药物不良反应和药物间的作用、解码 DNA；
- 提高运动成绩，减少运动伤害；

- 优化机器和设备性能，如基于历史数据和即时交通流量数据的智能交通管理，以及基于使用频率和传感器数据分析的电梯运行维护；
- 改善公共安全和执法，如预防各类欺诈（电信诈骗和信用卡诈骗等）和网络攻击等；
- 改善城市生活（如利用社交网络和移动网络传播交通信息和天气），实现优化管理；
- 金融交易，如高频量化交易、参考社交媒体的反馈来确定买卖策略等。

大数据是在原始数据基础上的提炼，也是数据挖掘和业务能力的体现。物联网主要完成了数据收集的任务，而数据提炼是大数据分析的任务。有了大数据分析，才是物联网的数据价值所在。空有裸数据不处理并不能解决问题，反而会被大量数据所淹没，最终将导致无法承担数据的存储成本而无法利用其价值。

一般来说，传统企业升级互联网和展开互联网+战略，首先解决的是设备联网问题，其次考虑的是这种投资究竟能够为产品带来何种附加值，包括业务支撑、市场竞争、上下游关系、平台整合。笔者曾经接触过一个物流设备供应商，该企业推进物联网项目的目的很杂乱，大致如下所示：

- 利用大数据和历史数据了解设备利用率，并优化配置企业维护团队构成。如果设备不常用，那么运维团队可以小一些。
- 设备的使用率有可能和用户的某些 KPI 有关联，那么数据对客户来说也是有价值的。
- 某些设备（如生产线、门禁和电梯）的使用率和企业开工率有关，有一定财务价值，供应商通过获取此类信息可以推断出其客户的业务运行状态，可以用来将客户分级，避免应收账款等财务风险。
- 通过远程操纵来促进企业回款，不过这涉及了设备产权、管理权、安全分级以及商业道德问题。

这是来自第一线的需求。企业的潜意识里知道数据就是钱，并想把数据抓在自己手中，赚取更多价值。更多的客户其实根本来不及考虑这么多，只是竞争对手推进了，他们感受到了威胁，所以也需要推进物联网，并将数据积累起来。但是如何利用数据，对于他们和合作伙伴来说都是一个难题。从某个角度来看，大数据是“金矿”。而且和“金矿”一模一样的情况是，必须在巨量的垃圾数据中“沙里淘金”。

2016 年，中国电信上海分公司与开发者分享其每日的电信数据。数据主要涉及 5W：Who/Whom/When/Where/What，分别对应个人信息、行为信息、位置信息和需求信息。

中国电信上海分公司的数据量规模和种类如下：

- 每分钟 8 万条位置更新信息，这仅仅是徐家汇商圈产生的数据；
- 每小时近 300 万次移动电话呼叫；
- 每天 70~100TB 及 30 亿次互联网访问；
- 运营商传统数据——业务支撑系统 BOSS、企业管理系统以及 ERP/CRM；

- 运营商的其他数据——大规模信令系统也被认为是大数据；
- 互联网和移动网络——图片、文本、音频、视频等非结构化数据以及各种半结构化数据。

虽然数据量够大，但这也仅仅是一家电信运营商数据的匿名数据，而且数据类型不够丰富。不过，这些大量的原始数据如果配合其他数据，将会产生新的数据服务。虽然笔者只是外部观察者，对大数据分析也不算内行，但可以展望这些数据提炼后可能的应用：

- 通过 CDMA 网络的终端分布，从而估算出交通状况，如公共交通、汽车交通、轨道交通的实时情况。
- 通过 CDMA 网络的终端分布，可以估算出紧急状况（如突发事件等），优化网络配置。
- 通过统计流向不同电商的 IP 流量，可以推算出电商的周期性商业价值。由于现在的网站大多采用 HTTPS/AJAX 技术，因此运营商只能够得到流量数据。商品级别的大数据是电商平台的。
- 通过短信和 IP 电话平台大数据，可以估算出电信诈骗风险点。
- 通过账单系统的大数据，可以挖掘客户价值和流失原因。
- 通过 Wi-Fi 大数据，可以挖掘地理位置和 O2O 服务。

有兴趣进一步了解详情的读者，可以在百度文库中检索《电信运营商掘金数据价值的探索》。这里面列举了大数据时代电信运营商的思考。此外，中国移动通信研究院对此也有专项讨论。

未来依托于大数据分析的商业模式可能如下所示。

- **出租或出售数据**：收集、过滤并租售数据本身来获益，“数据即资产”。
- **出租或出售信息**：交付数据统计分析后的结论与特性信息，类似于咨询服务。
- **精准营销**：通过个人和群体行为分析，进行针对化的销售推广。
- **数据分析业务**：基于交易模型、行为模式的大数据分析，用于金融风控、贷款、信用度、企业现金流调查。
- **运营数据空间**：网络存储可以形成以用户数据为基础的聚合平台，产生新的数据和业务。
- **大数据处理**：将非结构化数据（如语音、视频、图形等）通过智能算法进行处理的业务。

接下来我们分别看看数据分析在各个领域中采用的平台和工具。

10.5.3　科学计算数据分析工具

Python 在数据统计分析领域的普及趋势和 Python 的胶水特性与易用性密切相关，也和几个数据分析的 Python 包有密切关系。

物联网虽然大多是时序信号，但是在医学和科学、工程领域相关的物联网应用中，不仅仅处理简单的时序信号，还需要采用频域的傅里叶分析或者介于两者之间的小波变换。典型的例

子就是船舶、交通工具和电梯的振动监测，需要时常做音频傅里叶分析，来做远程无损检测，以避免重大事故。工业、地质、军工等领域更加严重依赖于各种科学分析工具。

10.5.3.1 NumPy

NumPy 是 Python 的开源数值计算扩展包，可以用来存储和处理大型矩阵。NumPy 将 Python 变成了 MATLAB 的免费替代语言之一。NumPy 和数学矩阵运算包 SciPy 配合使用会更加方便。

NumPy 的特性包括：

- 强大的 *N* 维数组 Array 对象；
- 复杂的广播（broadcasting）函数；
- 整合 C/C++/Fortran 代码工具包；
- 线性代数、傅里叶变换、随机数产生器。

除了以上的科学计算目的，NumPy 还可以作为多维数组的高效通用容器，可以包含任意数据类型。这使得 NumPy 可以很快地与许多数据库进行无缝整合。

10.5.3.2 SciPy

SciPy 是一款易于使用的 Python 工具包。基于 NumPy，包括了统计、优化、整合、线性代数模块、傅里叶变换、信号和图像处理及常微分方程求解。

SciPy 按照不同科学计算领域分成不同的子包（subpackage），如表 10-1 所示。

表 10-1 SciPy 子包

子 包	描 述
cluster	聚类算法
constants	物理学和数学常数
fftpack	快速傅里叶变换
integrate	积分和常微分方程求解
interpolate	插值和平滑
io	输入/输出
linalg	线性代数
ndimage	多维图像处理
odr	正交距离回归
optimize	优化和求根
signal	信号处理
sparse	稀疏矩阵和例程
spatial	空间数据结构和算法

续表

子　包	描　述
special	特殊函数
stats	统计分布和函数
weave	C/C++ 整合

可以看出，SciPy 基于 NumPy 基础之上，实现了大量科学工程计算方法和统计学方法。

10.5.3.3　matplotlib

数据分析最终需要将可视化组件呈现给用户。matplotlib 是一个 Python 图形框架，类似于 MATLAB/R 语言，专门用于数据可视化。John Hunter（1968—2012）是 matplotlib 的创建者。因为 matplotlib 与科学计算结合紧密，所以在此先单独介绍。后面还会介绍专门的数据可视化软件包。

matplotlib 是一个 2D 的 Python 绘图库，可以生成出版质量的图形，并可以跨平台进行交互。matplotlib 可以在 Python 脚本、Python 和 IPython 交互环境、Web 服务器和桌面 GUI 中使用。虽然说 matplotlib 是 2D 库，但它实际上可以输出 3D 绘图。在如图 10-2 所示的例图中，最右侧的例图就是 3D 绘图。

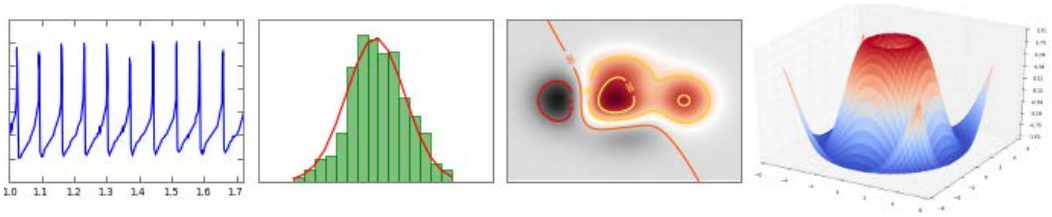


图 10-2　matplotlib 演示例图

pyplot 接口提供类似于 MATLAB 的接口进行绘图。尤其是和 IPython 交互环境整合时，更加容易。需要控制细节的用户可以通过面向对象接口或 MATLAB 风格的函数来控制线条风格、字体属性、轴线属性等。matplotlib 的截图、缩略图和例子可以分别在以下网址找到：

- http://matplotlib.org/users/screenshots.html
- http://matplotlib.org/gallery.html
- http://matplotlib.org/examples/index.html

例图中有 3D 绘图、金融 K 线图、对数图、极化图、数学表达式、TeX 渲染、手绘风格草图、EEG 脑电波图。其中 EEG 脑电波图是 matlibplot 与 pbrain 扩展配合的设计；pbrain 是 Python 语言的脑电波查看器，作者为 John Hunter。

缩略图中比较吸引人的有颜色图、图表着色、地形地貌图、雷达图、MRI 磁共振图、缩放

图、等高图，以及各类传统绘图，如柱状图（Bar）、折线图（Line）、区域图（Area）、饼图（pie）的 3D 版本。

10.5.3.4 FFT 频域分析实例

要完成这个快速傅里叶变换的例子，安装是个“拦路虎”：科学计算至少需要安装 matplotlib+NumPy+SciPy 三大软件包，以及底层依赖软件包。由于 Windows 下缺乏开源的 LAPACK/ATLAS 库，因此还需要 C/Fortran 编译器。SciPy 会遇到安装问题，好在有人已经提供了对应的解决方案。测试安装流程后发现，三大组件都有大量的依赖项没有自动化安装，同样需要手动安装许多组件。读者有兴趣可以通过安装 pip 来安装这三个组件，然后查看到对应依赖项的缺失，再利用 apt-get install 来安装所需要的组件。

1．Windows

在 Windows 下可以使用 Anaconda 安装包，它可以替我们一次安装好所需要的所有软件包。当然，我们也可以自己手动安装。但是 Windows 操作系统手动安装 SciPy，需要先安装 VC++ for Python 2.7。

7.1.3.2 节提到美国加州大学欧文分校的美国国立卫生研究中心的学者 Christoph Gohike，专门为 Windows 下的各类难搞的 Python 扩展包提供了 Python 预编译库，而且针对 32/64 位 Windows 不同版本 Python 2/3 分别提供支持。

网址如下：http://www.lfd.uci.edu/~gohlke/pythonlibs/#scipy。

读者可下载与系统相匹配的 SciPy 下载版本，如 scipy-0.17.1-cp27-cp27m-win_amd64.whl。

使用 pip 进行本地安装：

```
pip install scipy-0.17.1-cp27-cp27m-win_amd64.whl
```

可以解决 Windows 7 的 SciPy 安装问题。至于其他 NumPy/Matplotlib，可以使用 pip 安装。

2．Linux

Linux 下的安装方式如下：

```
sudo pip install numpy
sudo apt-get install libpng-dev
sudo apt-get install libfreetype6-dev
sudo pip install matplotlib
```

安装 SciPy 需要大量依赖项和细节，推荐使用：

```
sudo apt-get install gfortran libopenblas-dev liblapack-dev
sudo pip install scipy
```

满屏的字符串和漫长的编译过程，让笔者联想起构建 Linux 内核的过程。

下面是取自 NumPy 的教程源码：

```
>>> import numpy as np

>>> np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
array([ -3.44509285e-16 +1.14423775e-17j,
        8.00000000e+00 -5.68502218e-15j,
        2.33486982e-16 +1.22464680e-16j,
        1.44328993e-15 +1.77635684e-15j,
        9.95799250e-17 +2.33486982e-16j,
        0.00000000e+00 +1.64244978e-15j,
        1.14423775e-17 +1.22464680e-16j,  -1.44328993e-15 +1.77635684e-15j])
>>> import matplotlib.pyplot as plt
>>> t = np.arange(256)
>>> sp = np.fft.fft(np.sin(t))
>>> freq = np.fft.fftfreq(t.shape[-1])
>>> plt.plot(freq, sp.real, freq, sp.imag)
[<matplotlib.lines.Line2D object at 0x0000000009E71A58>, <matplotlib.lines.Line2
D object at 0x0000000009E71C18>]
>>> plt.show()
```

图 10-3 是运行程序后，从弹出视窗中保存下来的 matplotlib FFT 绘图。此外，利用波形文件来做测试是个很典型的例子。

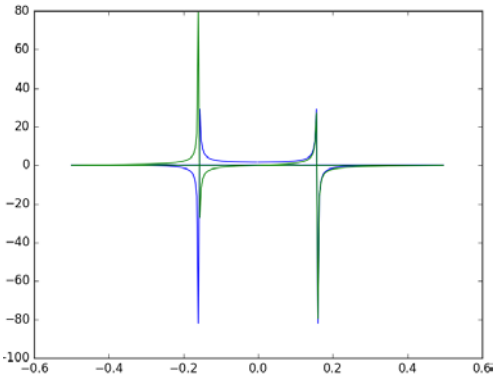


图 10-3　matplotlib FFT 绘图

先安装 PyAudio、PyLab，然后录制一段 wav 文件（Windows 下的非压缩文件），或者将某些 MP3 转回 wav 文件。 wave_demo.py 如下：

```
import numpy as np
import wave
wvf = wave.open("myaudio.wav","rb") #your wave file
params = wvf.getparams()
nchannels, sampwidth, framerate, nframes = params[:4]
str_data = wf.readframes(nframes)
wf.close()
```

```
#convert wave into array
s = np.fromstring(str_data, dtype=np.short)
#wave_data
s = np.fft.fft(s)  #fft
```

接下来，读者可以自行绘图；也可以将采集到的物理量数据通过 np.fromstring 函数进行 FFT 转换，看看其频域特性。

10.5.3.5 小波变换与 PyWavelets

傅里叶变换是很强大的工具，但是其分析的基础是周期性信号。对于大自然中普遍存在的非平稳信号，傅里叶变换有很大局限性。小波变换则可以不光观察到频谱，还可以定位该频谱的时间。关于两种换化的理解需要读者温习高等数学，不过知乎上有一篇文章《能不能通俗的讲解下傅里叶分析和小波分析的关系》图文并茂地介绍了小波变换的由来和优势。此外，知乎上有专门的“小波分析”话题，汇集了不少的相关问答。

Python PyWavelets 支持以下的小波变换：

- 1D 和 2D 正向和逆向离散小波变换（DWT/IDWT）；
- 1D 和 2D 平稳小波变换，即非抽样的小波变换；
- 1D 和 2D 小波包分解与重构；
- 计算小波逼近值和缩放函数；
- 七十多个内置小波滤波器，支持自定义小波；
- 单精度和双精度计算；
- 结果与 MATLAB Wavelet Toolbox 兼容。

10.5.3.6 PythonMagick

ImageMagick 是非常著名的图像处理工具。其 Python 被封装为 PythonMagick。或者应该这么说，PythonMagick 是 GraphicsMagick C++库的封装。而 GraphicsMagick 是从 ImageMagick 分支出来的软件。由于 ImageMagick/GraphicsMagick 是采用 C/C++编写的，所以 PythonMagick 处理图像也非常快，甚至可以实现批量处理。如果读者有兴趣，也可以采用 os 模块调用 ImageMagick 的命令行程序来实现图像处理。

笔者之所以介绍 PythonMagick/ImageMagick，是因为 ImageMagick 有专门针对图像的 DSP 处理算法，从数学上理解这就是一种 2D 的傅里叶变换。ImageMagick 的 convert 命令中有–fft 选项，专门用于离散傅里叶变换。

笔者曾经被每位同事拜托处理一张其家人的旧照片。处于保护个人隐私的需要，本书不方便提供其原照。但是许多读者可能知道 20 世纪 50 年代的照片冲印中有一种亚光片，即照片表面不是光滑的，而是一种凹凸状的表面。此类照片在扫描后，其表面在扫描仪的灯光反射下会

非常显眼，而且普通方式无法去除。网络上有许多 Photoshop 教程，其大多采用模糊滤镜，这样修图后凹凸点减弱，但却损失了照片图像的锐度，致使照片变得模糊。正确的方式是对其进行傅里叶变换，在频域中进行处理。

由于这种凹凸表面非常有规律，在变换后的 2D 频谱分布图中会很明显地出现一个峰值分量。可以在变换后的频谱分布图中直接针对峰值进行限值处理，相对于在图像频域中进行一次 2D 矩形滤波；或者采用其他类型的 2D 过滤器。过滤后的频谱再变换回 2D 时域信号。这时候再观察处理后的照片，这类凹凸表面被淹没在了人像信息中。这实现了在不损失照相原有信息的前提下，过滤掉表面凹凸噪点的方法。

通常笔者采用 GIMP 和相关插件来转换频域，手动抹去凹凸点的峰值。但是如果要实现自动化的图像修复，则需要利用程序，在 FFT 转换后的二维频谱分布图中自动提取特征值，定位峰值所在位置，并构建对应的滤波器。这需要使用 SciPy 和 PythonMagick 来配合实现算法。

10.5.3.7　GIMP

GIMP 是 ImageMagick 之外另一个著名的开源图像处理工具，与 ImageMagick 不同，GIMP 带复杂的 GUI 设计，所以其经常被拿来与 Photoshop 做比较。GIMP 的 UI 设计衍生出 PyGTK+ 项目，所以该软件包天生与 Python 有亲缘关系。

GIMP 支持插件设计，可以使用 Perl/Python/Script-Fusion（*.scm）插件。所以理论上说，上述 2D FFT 算法应该也可以通过插件形式提供。

关于 GIMP 的插件设计，请参考其官网：

http://registry.gimp.org/node/28124

10.5.3.8　PyCUDA 与高性能计算

著名显卡和移动方案供应商英伟达（NVIDIA）的 CUDA 是许多高性能计算的平台。其中，我们比较熟悉的是 Bitcoin 挖矿机。因为 Bitcoin 需要大量并行计算，所以 GPU 计算成为理想的计算平台。

NVIDIA 为 CUDA 计算引入了 PyCUDA，日益壮大的 Python 语言可以利用 CUDA 并行编程模型，在各种 HPC 高性能计算、大数据分析应用程序中充分利用 GPU 加速。实际上，这个软件包充分利用了 LLVM、CUDA 并行计算两种技术，其成为一种全新的计算模式。

笔者个人对 HPC 不太了解，也没有一手经验，只是看到朋友在做 3D 扫描的应用，采购的是 NVIDIA 的 Jetson 开发板，进行 GPU 编程。而且在初期验证算法的时候，其是利用 Amazon 的 GPU HPC 服务器进行论证的。Jetson 采用 ARM Cortex-A 的处理器，运行完整版 Ubuntu for ARM 系统。这也是嵌入式开发，是实时图形图像和并行处理的发展方向之一。传统的嵌入式图形图像更多采用 TI/ADI 的 DSP；或者 Xilinux/Altera 的 FPGA，采用 GPU 的开发可以实现原型开发，并快速投产。

笔者在采购笔记本电脑时，也偏爱带 NVIDIA 独立显卡的品种。不是为了游戏，而是为了有机会评估 CUDA/PyCUDA。后面提到的一些可视化软件包中采用 OpenGL 进行并行计算和图形渲染，也是一种值得关注的技术。

10.5.3.9 PyOpenCL

OpenCL（Open Computing Language，开放运算语言）是第一个面向异构系统通用目的并行编程的开放式、免费标准，也是一个统一的编程环境，便于软件开发人员为高性能计算服务器、桌面计算系统、手持设备编写高效、轻便的代码，而且其广泛适用于多核心处理器（CPU）、图形处理器（GPU）、Cell 类型架构以及数字信号处理器（DSP）等其他并行处理器，在游戏、娱乐、科研、医疗等各种领域都有广阔的发展前景。

利用 Python 的简单学习、功能强大的特点，我们可以利用 PyOpenCL 实现以往 OpenCL 才可以实现的并行编程等。Intel 的 CPU 虽然在并行核心数上无法匹敌 GPU 计算，但是合并 Altera 之后，CPU+FPGA 的架构或许可以充分利用 OpenCL 的能力。不过其工程进展尚待观察。

CUDA/OpenCL 不仅仅是并行处理的利器，还可以用于数据分析，其也是机器学习的技术平台之一。

10.5.4 统计学数据分析工具

Python 中的数据分析和科学计算相关的软件包如 NumPy、SciPy、pandas 都支持并包括了统计学工具在内，所以这两者有一定的重叠性。此外，还有一个专门的 Statmodels 软件包可用于统计学目的。

Statmodels

Statmodels 支持用户浏览数据、预估统计模型，并进行统计检验，还为供不同类型的数据和估值器提供广泛的描述性统计、统计检验、绘制函数和结果统计。Statsmodels 完全满足各种统计计算和数据分析。它的功能包括：

- 线性回归模型；
- 广义线性模型；
- 离散选择模型；
- 鲁棒线性模型；
- 时间序列相关建模与函数；
- 非参数回归估计；
- 各种演示用数据集；
- 各种各样的统计检验；

- 输入/输出工具可以制作大量文本、LaTeX、HTML 表格，在 NumPy/pandas 中读取 Stata 文件；
- 绘制函数；
- 单元测试以确保结果的正确性；
- 包含许多数据模型和扩展。

10.5.5　金融数据分析工具

金融行业的数据建模和量化投资是数据分析的重点领域之一。金融交易数据大多是时序数据，通过程序来建立数学模型和交易策略，进行回归测试分析，以验证模型和交易策略的正确性、最大回撤和收益率。

金融分析本身不能够算是纯粹的“物联网”应用，至少金融分析的数据来源和分析结论无法直接来自或者作用于智能联网设备。或许在今后的金融应用中，银行、投行、券商的各种用户终端，如 POS 机、ATM、外汇兑换点、移动 APP 可以更加智能化。比如，可以直接提供交易数据，或利用这些金融数据参与业务交易。除此之外，金融行业的数据分析主要用于风险控制等场景。

下面介绍金融分析所利用的 Python 软件包，主要是 pandas、NumPy、SciPy、matplotlib 和 scikit-learn 之类的工具集。上述软件包和物联网中的科学计算数据分析的工具集是类似的。这是因为金融和物联网数据都是时序信号，其工具和方法是可以共享的。这些软件包也可以用于物联网的大数据离线数据分析任务。

金融行业里的数据是某个交易品种的日内交易，包括不同时间颗粒的开盘价、收盘价、均价、成交量等。物联网收集的数据则是某台设备的一组物理量，包括不同时间颗粒的流量或者压力的峰值、谷值、均值等。虽然关注的点不同，但是其数据属性非常类似。所以，这些分析工具都能够共享使用。

实际上，金融行业的数据分析和分析结果的变现程度是非常高的。通过观察金融行业的数据交易和变现模式，可以对日后物联网数据交易的商业化有所借鉴。

10.5.5.1　pandas

Python Data Analysis Library，又称 pandas，是基于 NumPy 的一种工具。该工具专为解决数据分析任务而设计。pandas 纳入了大量库和标准的数据模型，提供了高效操纵大型数据集所需的工具，其中包括稀疏数据。pandas 提供了快捷处理数据的方法，主要用于金融交易的时序数据。它使得 Python 成为一种强大的高效数据分析工具。

pandas 支持的数据结构如下。

- 一维数组：Series，类似于 NumPy 中的 array，与 Python list 不同，array 和 Series 的数

据类型必须一致。其中，Time-Series 时间序列可以用来描述常见的金融交易数据。

- 二维数组：DataFrame，与 R 语言中的 data.frame 类似，可以将 DataFrame 理解为 Series 的容器。金融分析往往围绕 DataFrame 展开。
- 三维数组：Panel，可视为 DataFrame 的容器。

10.5.5.2 通联数据

通联数据是在线量化交易平台方面的领军企业。其设计的“优矿”在线平台是私人量化研究与交易云平台，旨在打破金融量化的壁垒，为投行、私募股权、基金、学者提供华尔街专业量化机构的装备。优矿提供了高质量的海量金融数据与高性能分析工具，与用户共享智慧与金融在大数据时代的红利。

通联数据文档表明：优矿基于一系列重要的数据分析包：

- pandas，数据分析软件包；
- NumPy，数据容器基础包；
- SciPy，科学计算软件包；
- matplotlib 和 seaborn，数据可视化软件包；
- sklearn，机器学习软件包。

通联数据提供大量的金融数据 API，如下所示。

- 股票：沪深股市行情、国内外主要期货合约，以及指数行情信息；沪深股票的基本信息和 IPO、配股、分红、拆股、股改、行业及回报率等信息。
- 报表：沪深上市公司披露的 2007 年以来的所有三大财务报表数据，以及业绩预告和快报数据信息。
- 债券：债券基本信息和发行上市、付息、利率、评级和评级变动、债券发行人评级及变动、担保人评级及变动等信息。
- 基金：基金基本信息和资产配置，以及每日净值、收益情况、净值调整等信息。
- 期货：中国四大期货交易所期货合约的基本要素信息和国债期货的转换因子信息。
- 指数：指数基本要素信息和成分构成情况、国内外指数的成分股权重情况，以及中证指数信息和指数成分股的自由流通股与权重信息。
- 宏观：中国及全球宏观指标和数据、行业经济指标和数据，以及中国香港交易所股票的基本信息和上市公司行为等信息。
- 期权：期权合约的基本信息、每日盘前静态数据和期权标的物信息。

其他特色数据如下：

- 量化因子数据；
- 雪球、股吧等社交媒体数据；

- 主流媒体新闻数据；
- 股票主题数据；
- 淘宝、天猫等电商数据；
- 其他第三方供应商的数据，包括恒生聚源、申万、朝阳永续、汤森路透、中诚信资讯等机构。

可以这么说，即使读者无意于金融量化投资，通联优矿也是一个不容错过的数据分析实例，它为用户体验大数据分析提供了最直接的平台和数据来源。在知乎上还有 vn.py 项目也是值得读者跟踪的金融分析交易项目，其同样采用 Python 编写。

10.5.6　大数据平台与生态

大数据平台的发展非常快。从离线批处理的 Hadoop，到实时处理的 Storm，再到兼顾批处理和实时处理的 Lambda，让人眼花缭乱。

10.5.6.1　Apache Hadoop

云计算和大数据说来说去绕不过谷歌，而 Hadoop 是谷歌大数据理论中 MapReduce 的开源实现。Hadoop 本身是用 Java 语言编写的。所以，Python 与 Hadoop 之间需要通过 Jython 进行。此外，Hadoop 的 Streaming API 通过标准输入/输出在 Map 和 Reduce 之间传输数据。所以，Streaming API 实际上可以采用任何语言进行编程，包括 Python（CPython）。

1．Hadoop Jython

Hadoop 最经常使用的例子是 Wordcount，即统计一篇文章中所有文字的重复出现频率。Hadoop 文档提供了 Java 和 Python 例子，Python wordcount 例子运行于 Jython 之上。

2．Hadoop Streaming

Hadoop Streaming 是采用 Java 之外的语言实现的 mapper 和 reducer 程序，从而利用 Hadoop 并行计算能力进行大数据分析。其特点是利用 UNIX 操作系统的标准输入/输出完成与 Hadoop 平台的通信。所以任何能够利用标准输入/输出的语言均可以利用 Hadoop Streaming，包括 Python。Hadoop Streaming 工具包是 hadoop-streaming-.jar。具体流程是采用 Python 编写 mapper/reducer 两个程序，并将这两个程序交由 hadoop-streaming-.jar 运行。

Python Streaming API 需要注意：

- 尽量使用迭代器（Iterator），避免在内存中保存 stdin 的输入；
- 采用 rstrip()来取出 stdin 结尾的'\n'；
- 需要手动切割字符串将键值分割开来；
- 可以使用 groupby 和 itemgetter 来获取键值列表。

随着数据容量、多样性和速度的增长，作为离线批处理框架的 Hadoop 无法满足实时分析

的要求。接下来我们介绍 Apache 基金会旗下的两个实时大数据分析产品：Storm 和 Spark。

10.5.6.2 Apache Storm

Twitter 于 2011 年收购了 BackType 公司，并将 Storm 提交给 GitHub。2014 年其成为 Apache 旗下的顶级项目之一。Storm 被誉为实时处理领域的 Hadoop。Storm 大大简化了面向庞大规模数据流的处理机制，从而在实时处理领域扮演着 Hadoop 在批量处理领域所扮演的角色。

Storm 项目采用 Clojure 编写，其设计目标在于将“流”（如输入流）与“栓”（处理与输出）结合在一起并构成一套有向无环图（DAG）拓扑结构。Storm 拓扑结构运行于集群之上。Storm 调度程序根据拓扑配置将处理任务分发给集群中的各个工作节点。尽管 Storm 本身基于 Clojure 并运行于 JVM 之上，但其流与栓可以通过所有语言进行编写，只要能够在标准输入/输出上使用 JSON 即可。尽管 Storm 同时提供原语，可以实现通用目的的分布式 RPC，理论上这可以用于任何分布式计算任务，但是其根本优势在于事件流处理方面。

什么是 Clojure

作为当今最主流的运算平台 JVM，把函数式编程语言引入 JVM 也是很多人尝试的方向，Clojure 就是其中之一。Clojure 是一个在 JVM 平台运行的动态函数式编程语言，其语法接近于 LISP 语言。Clojure 源码编译为 Java 字节码，在 JVM 平台运行。

10.5.6.3 Apache Spark

Spark 由加州大学伯克利分校打造，2014 年其成为 Apache 的顶级项目之一。Spark 也支持面向流的处理机制，它也是具备通用性的分布式计算平台。

Spark 可以运行于 Hadoop 集群，但依赖于 YARN 的资源调度能力。Spark 由 Scala 编写，支持多语言编程，不过特殊 API 只支持 Scala、Java 和 Python。

什么是 Scala

Scala 是一门多范式的编程语言，支持面向对象、函数式编程和高层并发模型，并运行于 JVM 之上。

Storm/Spark 都面向实时处理。具体选用哪一种平台，需要用户根据自己的需求以及两个平台的约束条件来进行。表 10-2 是部分指标和需求的对比表格。

表 10-2 Storm/Spark 实时数据流处理对比

比较项目	胜出者
事件流处理，复杂事件处理	Storm
Hadoop 集群，图形处理，SQL 访问，批量处理	Spark
多语言支持能力	Storm
交互式 shell 实现数据分析探索	Spark

续表

比较项目	胜出者
面向原始操作的连续性方法调用	Storm
创建类，实现接口	Spark

最好对两套平台做一番验证性测试，来确定哪一种平台更适合自己的实时分析平台。

10.5.6.4　Lambda

它是 Storm 作者 Nathan Marz 推出的实时大数据处理框架，整合了离线批处理计算和实时计算。他认为一个大数据系统必须具备的属性包括：

- 健壮性和容错性（Robustness and Fault Tolerance）；
- 低延迟的读与更新（Low Latency Reads and Updates）；
- 可伸缩性（Scalability）；
- 通用性（Generalization）；
- 可扩展性（Extensibility）；
- 内置查询（Ad-hoc Query）；
- 维护最小（Minimal Maintenance）；
- 可调试性（Debuggability）。

Nathan 认为所谓“数据系统”的定义如下：

```
Query = function(all_data)
```

Nathan 的 Lambda 架构分为三层：Speed/Serving/Batch，并利用以下三个等式来描述 Lambda 架构。

```
batch_view = function(all_data)
realtime_view = function(realtime_view, new_data)
query = function(batch_view, realtime_view)
```

不愧是采用函数式编程的高手，连系统架构都用函数表达。Lambda 官网网址如下：http://lambda-architecture.net/

10.6　数据可视化

数据可视化是数据分析的重要手段，是决策支持与用户的接口。绘制一个吸引人注意的图表不仅仅是取悦用户，也是为用户提供直观分析结果的途径。在与观众或用户交流观点时，可视化同样是沟通的基础。所以，非常有必要让图表吸引用户注意力，并努力通过图表将意见表

达出来，继而影响用户的决策。数据过于生硬，以图表的形式表达，还可以获得附加分析结论。

数据可视化是数据经过收集、整理、统计、分析之后，提供给用户做决策的重要依据。由于属于人机接口的一部分，因此其技术迭代和候选方案特别多。

10.6.1 数据可视化的发展趋势

数据可视化，是关于数据视觉表现形式的科学技术研究。数据的视觉表现形式被定义为，以某种概要形式抽提出来的信息，包括相应信息单位的各种属性和变量。

它又是一个处于不断演变之中的概念，其边界在不断地扩大。其主要指的是采用较为先进的技术方式充分利用图形、图像处理、计算机视觉、用户界面，通过表达、建模以及各种立体、表面、属性和动画显示技术，对数据加以可视化解释。与立体建模之类的特殊技术方法相比，数据可视化所涵盖的技术方法要广泛得多。

关于数据可视化，笔者推荐几本参考书：

- 《数据可视化之美》（*Beautiful Visualization*，作者为 Julie Steele、Noah Iliinsky，O'Reilly 出版社出版）。该书采用了大量彩页，可以让工程师开阔一下视野，学会从市场、管理层和用户角度来理解大数据分析和可视化的价值所在。
- *Beginning Python Visualization*（作者为 Shai Vaingast，Apress 出版社出版）。该书专门论述 Python 在数据可视化中的各类应用和技术。
- 《Python 科学计算》，作者为张若愚（HYRY Studio），该书提供免费下载的电子书和网页。它是很棒的可视化学习资源。

根据笔者的观察，数据可视化作为单独归类的技术门类，其变化趋势是，交互化、大数据、多形态、在线化、目录化（以用户直接使用的目录形式提供）、3D 化、动态化、网络化（复杂网络关系与图论）。为此，笔者收集了与 Python 有关联的可视化扩展包，并针对不同特性做了一张对比表，如表 10-3 所示。

表 10-3　Python 与数据可视化

名　　称	交互性	大数据	多形态	在线化	目录化	3D	动态化	网络化
matplotlib	支持		支持	支持		支持	支持	
seaborn	支持		支持	支持		支持	支持	
ggplot	支持		支持		支持			
Pygal	支持		SVG	支持	支持		支持	
Plotly	JavaScript	支持	支持	支持	支持	支持	支持	
TVTK	支持		支持		支持	支持		
Folium	支持	支持	GIS	支持				

续表

名　　称	交互性	大数据	多形态	在线化	目录化	3D	动态化	网络化
NetworkX	支持	支持		支持	支持	支持	支持	支持
mpld3	D3.js	支持	支持	支持	支持	支持	支持	
Bokeh	D3.js	支持	支持	支持	支持	支持	支持	
VPython	支持		支持			支持	视频	
Chaco	支持							
MoviePy						支持	视频	
Mayavi	支持		支持			支持	视频	
Vispy	支持	支持	支持			支持	视频	

注意　因无法针对每种软件包做仔细深入的评估，表 10-3 仅供参考。

10.6.2　matplotlib

在 Python 中，有许多数据可视化途径。但 Python 的可视化技术起源于 matplotlib。该软件包功能强大，但是略显复杂。许多 Python 包再次封装了 matplotlib，提供简化的方式帮助用户实现数据可视化。

10.6.3　seaborn

seaborn 是一种 Python 库，用来制作外观华丽、数据充实的统计图表。它基于 matplotlib，并紧密结合 PyData，支持 NumPy/pandas 数据结构，并包含了 scipy/statsmodels 的统计学例程。图 10-4 是 seaborn 的若干种例图。

matplotlib 的自动化程度非常高，但掌握系统设置来设计一个美观的图表还是很困难的。为了控制 matplotlib 图表的外观，seaborn 模块自带许多定制的主题和高级接口。

seaborn 的功能如下：

- 基于 matplotlib 默认外观基础上的内置主题；
- 调色板工具可以制作美观的绘图，并揭示数据中的潜在模式；
- 用于数据子集间的比较的一元和二元分布可视化函数；
- 用于独立变量和从属变量的线性回归模型的适配工具与可视化工具；
- 用于发现矩阵结构的数据可视化和聚类算法；
- 绘制统计时间序列数据函数及估值；
- 构建绘图网格的高级抽象算法，可轻松构建复杂可视化效果。

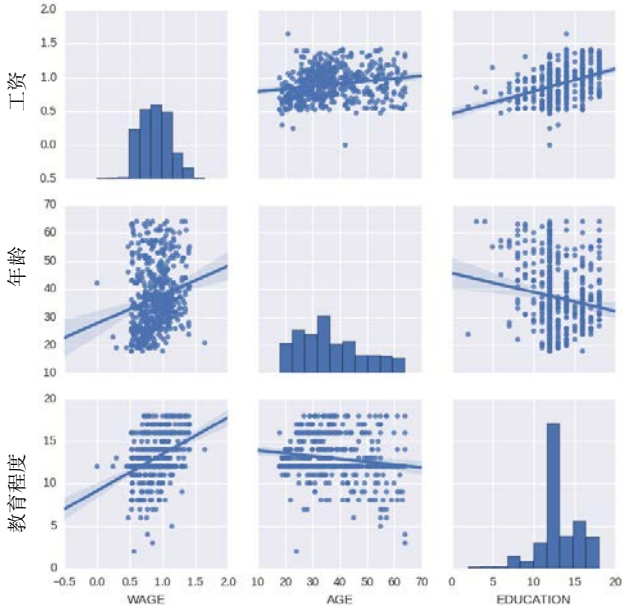


图 10-4　seaborn 演示例图

seaborn 的目的是让数据可视化成为探索和理解数据的核心。基于数据帧和数组的绘图函数包含整个数据集，进行必要的回归和统计模型计算，可产生信息丰富的绘图。seaborn 的设计目的和 R 语言的 ggplot 类似，但其采用了命令式和面向对象风格，试图简化构建复杂绘图的过程。seaborn 基于 matplotlib，但是易用性比 matplotlibt 更好。

seaborn 网站是斯坦福大学教育网站的一部分。其可以使用多种方式安装：

```
pip install seaborn
pip install git+git://github.com/mwaskom/seaborn.git#egg=seaborn
```

在 seaborn 网站的 Gallery 栏目中，罗列了各种绘图类型。读者可在其官网中仔细寻找合适的绘图类型。

10.6.4　mpld3

matplotlib/seaborn 一般用于桌面应用软件。浏览器中的绘图引擎则以 D3.js 为主。在 Web 前端中，D3.js 应用非常广泛，成为主流数据可视化工具之一。D3.js 的全称为 Data Driven Documents，由《纽约时报》的 Mike Bostock 和斯坦福大学的同仁一起合作开发。

mpld3，即 matplotlib+D3.js，其将流行的 matplotlib 和 D3.js 整合在一起，这可让开发者使用简单的 API 将 matplotlib 绘图导出到 HTML 代码中。我们可以在标准网页、博客和 IPython

笔记本工具中使用这些绘图。同时，它还为这些绘图增添了交互动作。其安装非常简单。

```
pip install mpld3
```

图 10-5 是交互式的 HTML Tooltip 演示，当鼠标停留在某一点上时，会提示数据属性。由于截屏时，系统会将鼠标隐去，所以图 10-5 中看不到鼠标。

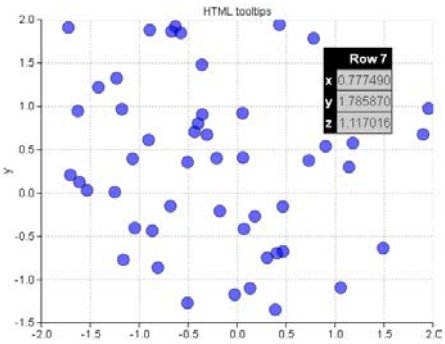


图 10-5　mpld3 的 HTML Tooltip 演示截图

10.6.5　Chaco

Chaco 是 Enthought 科学计算软件包中关于数据可视化的一部分。可以采用多种方式安装 Chaco，但是使用 Enthought Python Distribution（EPD）是最简单的方式。图 10-6 是其若干例图集锦。实际上，Enthought 还提供了许多其他软件，安装 EPD 可以节省大量时间。EPD 新版的名称为 Canopy，它是商业收费软件。我们可以找一下其早期版本做些评估。

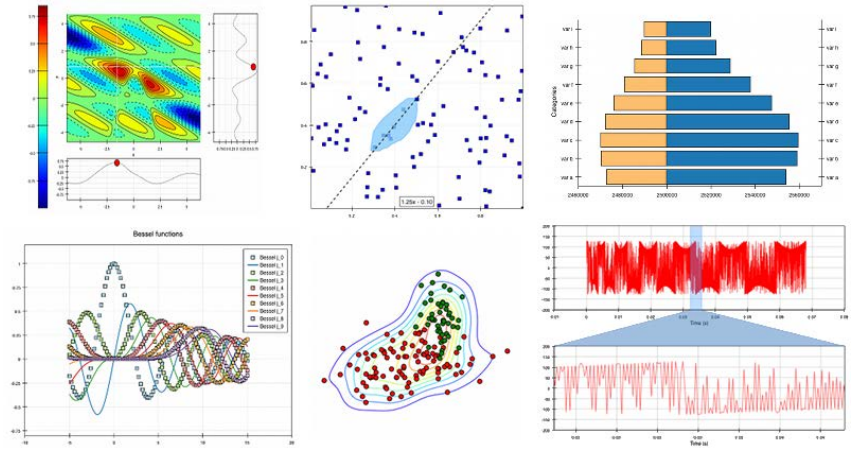


图 10-6　Chaco 数据可视化包例图

在 Chaco 的例子程序中，有一个音频频谱分析示例很吸引人。如图 10-7 所示，它不仅可以显示音频信号的波形、频谱，还可以显示频谱随着时间演变的过程。

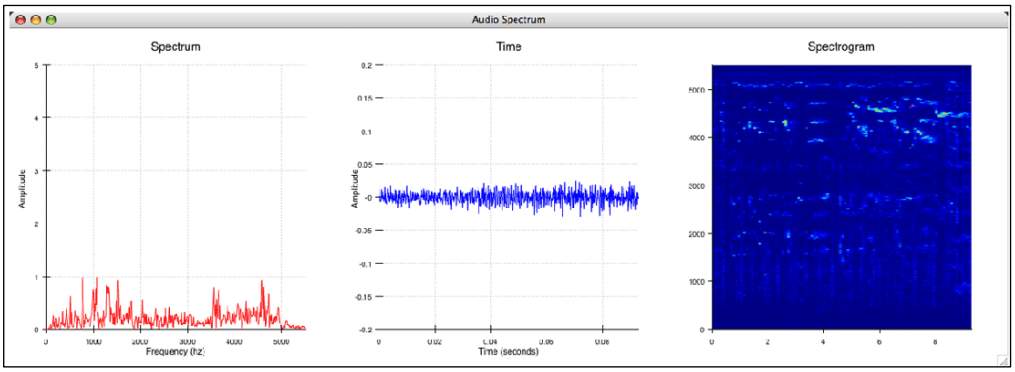


图 10-7　Chaco 音频分析可视化例图

10.6.6　Pygal

Pygal 是一个开源程序，用于创建向量图形。使用 pip 安装：

```
pip install pygal
```

Pygal 依赖于 lxml 加速渲染速度，使用 cairosvg、tinycss、cssselect 渲染 PNG 图像。其显示效果如图 10-8 所示。

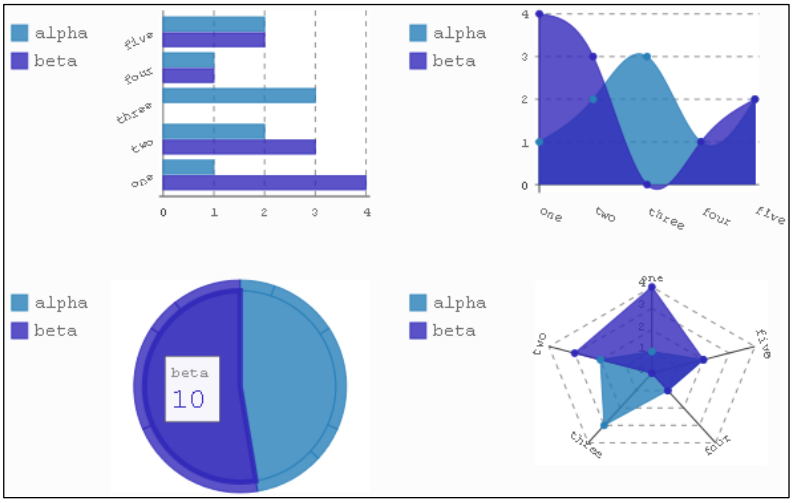


图 10-8　Pygal 可视化演示例图

```
import pygal
pygal.Bar().add('1', [1, 3, 3, 7]).add('2', [1, 6, 6, 4]).render()
```

以上 render()的结果是返回一个 SVG 文件对象。所以，在终端上是看不到任何东西的；必须将其放在 Web 服务器或者专门的 SVG 查看器查看实际的效果。

10.6.6.1　Pygal 输出

Pygal 可以输出 SVG（包括字符串）、PNG、XML/lxml Etree、Base64 URI、浏览器、PyQuery、Flash WebApp、Django 的 HTTP 响应。上面的代码可以输出为 SVG，或者直接输出到默认浏览器。

```
import pygal
pygal.Bar().add('1', [1, 3, 3, 7]).add('2', [1, 6, 6, 4]).render_to_file('demo.svg')
pygal.Bar().add('1', [1, 3, 3, 7]).add('2', [1, 6, 6, 4]).render_to_browser()
```

读者可自行运行一次。

10.6.6.2　SVG 简介

SVG 是一种基于 XML 的二维图形描述语言。SVG 兼容三种图形对象：矢量图形、图片、文本。图形对象可以组合分组、添加样式和变形处理。这些特性设置包括嵌套变形、剪切路径、alpha 通道蒙版、滤镜效果与模板对象。SVG 图形具备交互性和动态性。SVG 的动画效果是通过动画元素或动作脚本来实现的。

基于 SVG 的特性，可以用于创建地图和 LBS 应用。详情请参见本章延伸阅读部分。

10.6.6.3　嵌入 HTML 网页

在浏览器中使用 SVG 可以有两种方式：使用 embed Tag，或者将 SVG 内容嵌入 HTML。

embed Tag：

```
<figure>
  <embed type="image/svg+xml" src="/mysvg.svg" />
</figure>
```

直接嵌入 HTML：

```
<figure>
  <!-- Pygal render() result: -->
  <svg
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns="http://www.w3.org/2000/svg"
    id="chart-e6700c90-7a2b-4602-961c-83ccf5e59204"
    class="pygal-chart"
    viewBox="0 0 800 600">
    <!--Generated with pygal 1.0.0 ©Kozea 2011-2013 on 2013-06-25-->
    <!--http://pygal.org-->
    <!--http://github.com/Kozea/pygal-->
```

```
    <defs>
      <!-- ... -->
    </defs>
    <title>Pygal</title>
    <g class="graph bar-graph vertical">
      <!-- ... -->
    </g>
  </svg>
  <!-- End of Pygal render() result: -->
</figure>
```

10.6.6.4 局限性

有使用者反映 Pygal 有两个局限性：

- 在同一个图表中，似乎没有办法混合不同类型的图形，例如线形图和直方图。
- 与其他免费软件间的互操作性和兼容性仍然有缺陷。

具体情况需要读者自行评估。

10.6.7 Plotly

这是数据可视化平台 Plotly 推出的同名产品，作为强大的开源 JavaScript 绘图库，其支持多种不同类型的图表，包括地图、箱形图和密度图，以及更常见的条状图和线形图等许多图形。其显示效果如图 10-9 所示。最新版本的 Plotly.js 可以免费地用于任何项目，其源代码已发布在 GitHub。

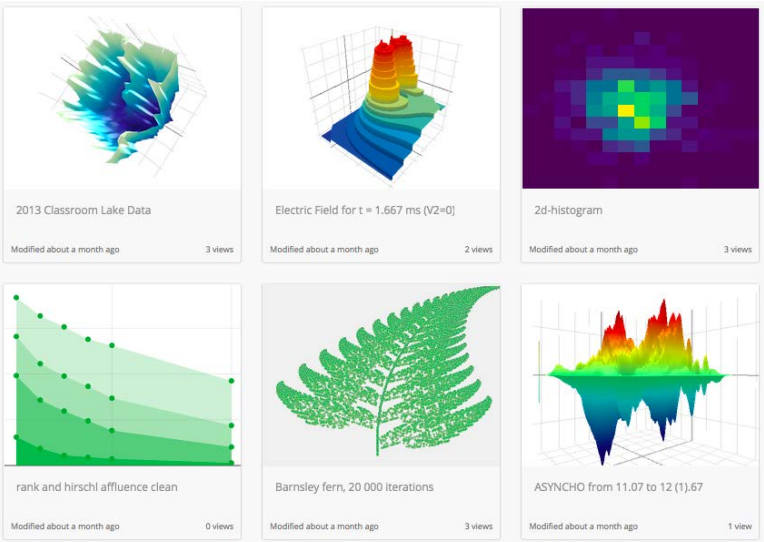


图 10-9 Plotly 演示例图

此前，Plotly 在 R、Python、MATLAB 的客户端中一直是开源的，但核心图表层 Plotly.js 却是闭源的。开源后，Plotly 可以 100%离线地用于 R Studio、MATLAB 或 Python Jupyter 环境中。

Plotly 声称：和许多 JavaScript 可视化库不同，Plotly.js 不依赖于 jQuery，而是基于新的开源 JSON schema。这使得 Plotly.js 的性能显著超过其他竞品。由于采用了 JSON 图表规范来实现交互性可视化，因此数据格式转换将变得更加简单，比如将 CSV 文件转成 Excel 图表、Python 代码、交互图表以及 R 代码。

Plotly 图形库利用 Python 构建交互式出版质量的在线图形。在其官方教程中，支持线条图（line）、散列图（scatter）、面积图（area）、条形图（bar）、方差条形图（error）、箱形图（box）、直方图（histogram）、热力图（heatmap）、嵌套图（subplot）、多轴图（multiple-axes）、极化图（polar）和泡泡图（bubble）。

Plotly 提供的是商业托管服务，其也有受限的免费服务。从绘图效果看，它提供的绘图质量很高。

10.6.8　TVTK

VTK 是一套开源的三维数据可视化工具，主要用于三维计算机图形学、图像处理和可视化。VTK 还是在 OpenGL 基础上采用面向对象原理进行设计和实现的。它由 C++编写，包涵了 2000 个类用于处理和显示数据。不仅如此，VTK 也是一个开放源码、跨平台、支持并行处理高性能计算的函数库，其被广泛用于军事、工业、医疗、建筑、气象、生物学、航空航天领域。VTK 最初的设计者均来自 GE 集团研发部门。后成立 Kitware 公司专门开发 VTK。除了 VTK，Kitware 旗下还有许多其他产品，如 ITK、OpenChem 等。图 10-10 是用 VTK 绘制的 3D 等高图例图。

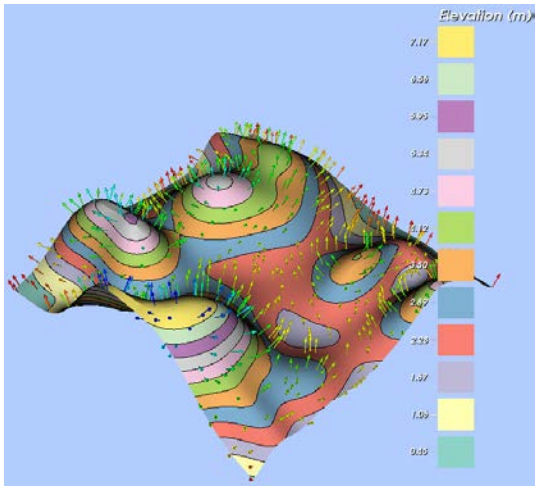


图 10-10　用 VTK 绘制的 3D 等高图例图

VTK 官网网址：http://www.vtk.org/。

VTK 在 Python 下有标准的绑定，不过其 API 和 C++相同，不能体现出 Python 作为动态语言的优势。TVTK 的全称是 Traited VTK。由 Enthought 开发，封装了标准 VTK 库，并提供了 Python 风格的 API、trait 属性和 NumPy 的多维数组。TVTK 是 Enthought 产品的一部分，需要使用 setup_tvtk.py 来安装。

```
$ cd tvtk
$ python setup_tvtk.py build_ext --inplace
$ cd tests
$ python test_tvtk.py
```

由于使用 TVTK 需要 VTK 的背景知识，请感兴趣的读者自行阅读其文档。VTK/TVTK 具备大数据的特点，分布式架构可以处理 PB 级别的数据。

10.6.9 VPython

图 10-11 为 3D 蜂窝球形图，采用 VPython 渲染，来自 4dsolutions.net。

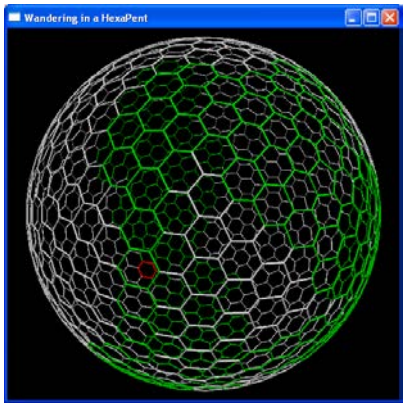


图 10-11　用 VPython 绘制的 3D 蜂窝球形图

VPython 是一套简单易用的三维图形库，使用它可以快速创建三维场景和动画。和 TVTK 相比，它更适合于创建交互式的三维场景，而 TVTK 则更适合于对数据进行三维可视化。

VPython 制作 3D 动画很简单：在程序循环中不断地修改场景中的各个模型及照相机属性就可以完成动画效果。其用户接口设备支持标准的键盘鼠标。但有理由相信，利用其他外部设备如 Android 设备、串口/蓝牙连接的 3D 传感器或其他高级输入设备来模拟标准输入设备，也可以操纵 VPython 动画。

VPython 需要下载对应的安装包。请到其官网下载 VPython 的安装软件进行安装。有一点让笔者有些惊奇：VPython 的原生环境是 Windows，在 Linux 下必须首先安装 WINE 虚拟机后

再运行 VPython 安装程序；然后还需要下载 python27.dll，以便在 Linux/WINE 中使用。笔者感觉 VPython 的安装方式有些凌乱，很难理解为何其不采用官网 PyPI 提供安装包。其最新的安装方式支持：

- GlowScript VPython 在线服务；
- Continuum Anaconda Python 3.2；
- Enthought Canopy Python 2.7；
- IPython Jupyter 模块。

VPython 让笔者联想到 Processing 3 中的 Geometry 3D 互动模型。两者的区别在于编程语言：Processing 使用 Java，VPython 使用 Python。Python 代码更轻省，就是希望能够统一安装包。

10.6.10　Folium

Folium 可以使用 Python 的强大生态系统来处理数据，然后用 Leaflet.js 地图库来展示数据。其与强大的谷歌地图/百度地图无异。Folium 内置来自 OpenStreetMap、MapQuest Open、MapQuest Open Aerial、Mapbox 和 Stamen 的地图元件（tilesets），并且支持用 Mapbox 或者 Cloudmade API keys 自定义地图元件。Folium 支持 GeoJSON 和 TopJSON 叠加（overlay），并绑定数据来创造一个分级统计图（Choropleth map）。

安装：

```
pip install folium
```

测试：

```
import folium
map_osm = folium.Map(location=[45.5236, -122.6750])
map_osm.create_map(path='osm.html')
stamen = folium.Map(location=[45.5236, -122.6750], tiles='Stamen Toner', \
   zoom_start=13)
stamen.create_map(path='stamen_toner.html')
```

以上代码会在当前文件夹中产生 osm.html 和 stamen_toner.html，其显示的是美国俄勒冈州波特兰市区地图。

笔者亲自测试的结果显示：国内用户可以访问 OpenStreeMap、MapRequest Open、Stamen、MapBox 的地图贴片。但由于各家网站大量采用了国外的 CDN 服务，所以无法加载 JavaScript 代码，导致地图贴片也无法加载。可惜的是，Folium 尚未支持国内的地图 API 和地图贴图。

Folium 可以通过百度地图、必应地图或者其他供应商 API 实现对国内地图的支持。对于许多网站后台常见的地图应用，如根据地理位置统计访客量和订单量的应用，也可以使用 SVG（Pygal）或者 D3.js（Bokeh）来实现。

10.6.11 NetworkX

NetworkX 是用 Python 语言编写的软件包，于 2002 年 5 月发布，便于用户创建、操作和学习各类复杂网络。这里的网络指的是以图论为基础的网络。

利用 NetworkX，可以采用标准化和非标准化的数据格式存储网络，生成多种随机网络和经典网络，并分析网络结构，建立网络模型，设计新的网络算法，进行网络绘制等。NetworkX 的特点如下：

- 针对图形、有向图、多重图而设计的数据结构；
- 多种标准图形算法；
- 网络结构和分析方法；
- 经典图形、随机图形和合成网络图形生成器；
- 节点（node）可以是任意对象，如文本、图形和 XML 记录；
- 边缘（edge）可以包含任意数据，如计权和时间序列；
- 采用开源 BSD 许可证；
- 测试完整，1800 个单元测试，90%代码覆盖率；
- 快速构建原型，易学，跨平台。

NetworkX 的使用者包括数学家、物理学家、生物学家、计算机学者和社会学家。对复杂网络科学的深入探讨，我们可以搜索 Albert、Newman、Mendes 等人的著作；对于图论，则可以搜索 Bolloas、Diestel 和 West 等人的著作。本书提到的思维导图也可以使用 NetworkX 来实现。

NetworkX 显示效果参见图 10-12。WSN 的网络拓扑、区块链技术，这些与网络节点有关的设计，均可以通过 NetworkX 进行可视化。传统网络仿真工具如下：

- ns-2/ns-3，GNS3；
- Cloonix；
- IMUNES；
- LINE；
- Marionnet；
- NetKit；
- Psimulator2；
- Shadow；
- UNL/UniNetLab；
- mininet（Python）；
- CORE（Python）。

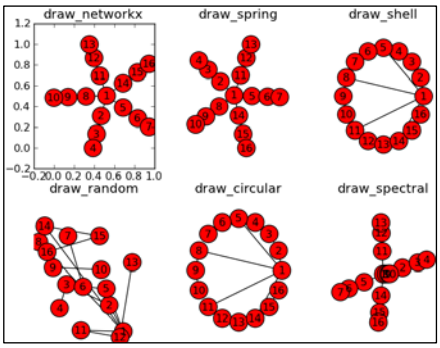


图 10-12　NetworkX 绘制的各种类型网络图例图

但与 WSN 协议设计与网络仿真有关的软件相对较少，NetSim 算是一个，也可以使用 MATLAB。

安装与使用

```
pip install networkx
>>> import networkx as nx
>>> G=nx.Graph()
>>> G.add_node("spam")
>>> G.add_edge(1,2)
>>> print(list(G.nodes()))
[1, 2, 'spam']
>>> print(list(G.edges()))
[(1, 2)]
```

10.6.12　Bokeh

Bokeh 的原意是背景虚化，这是摄影术语。在可视化中，Bokeh 是专门针对 Web 浏览器的交互式可视化 Python 库，这是 Bokeh 与其他库的核心区别。Python 的在线 Web 交互可视化必然走向与 JavaScript 结合的结果。Bokeh 的目标是充分利用 D3.js 的绘图风格提供优雅、简洁的图形，并扩展与超大数据集或者流式数据集之间的交互能力。Bokeh 可以用于快速构建交互绘图，控制仪表盘和数据应用。

10.6.12.1　安装

推荐使用 Anaconda Python 发行版安装 Bokeh。

```
conda install bokeh
```

或者通过 pip 安装（如果已经安装了 NumPy 的前提下）：

```
pip install bokeh
```

10.6.12.2 产生绘图

bokeh_demo.py：

```
from bokeh.plotting import figure, output_file, show
output_file("test.html")
p = figure()
p.line([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], line_width=2)
show(p)
```

在浏览器中查看运行代码后产生的 test.html，其显示效果如图 10-13 所示。

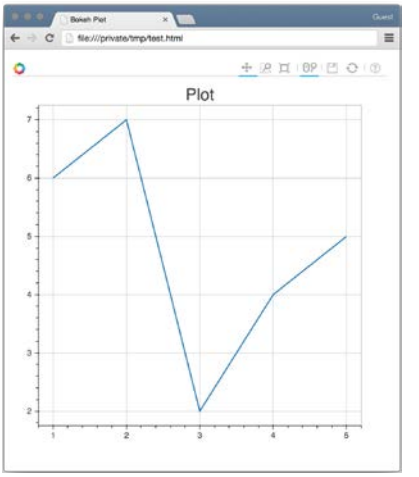


图 10-13 Bokeh 演示图例

10.6.12.3 更多实例

由于 D3.js 数据驱动的特性，因此 Bokeh 非常适合大数据可视化应用。

texas.py 如下：

```
from bokeh.models import HoverTool
from bokeh.palettes import Viridis6
from bokeh.plotting import figure, show, output_file, ColumnDataSource
from bokeh.sampledata.us_counties import data as counties
from bokeh.sampledata.unemployment import data as unemployment

counties = {
    code: county for code, county in counties.items() if county["state"] == "tx"
}

county_xs = [county["lons"] for county in counties.values()]
county_ys = [county["lats"] for county in counties.values()]
```

```
county_names = [county['name'] for county in counties.values()]
county_rates = [unemployment[county_id] for county_id in counties]
county_colors = [Viridis6[int(rate/3)] for rate in county_rates]

source = ColumnDataSource(data=dict(
    x=county_xs,
    y=county_ys,
    color=county_colors,
    name=county_names,
    rate=county_rates,
))

TOOLS="pan,wheel_zoom,box_zoom,reset,hover,save"

p = figure(title="Texas Unemployment 2009", tools=TOOLS,
           x_axis_location=None, y_axis_location=None)
p.grid.grid_line_color = None

p.patches('x', 'y', source=source,
          fill_color='color', fill_alpha=0.7,
          line_color="white", line_width=0.5)

hover = p.select_one(HoverTool)
hover.point_policy = "follow_mouse"
hover.tooltips = [
    ("Name", "@name"),
    ("Unemployment rate)", "@rate%"),
    ("(Long, Lat)", "($x, $y)"),
]

output_file("texas.html", title="texas.py example")

show(p)
```

以上例子读取样本数据的样本并产生对应的 texas.html。其绘图不仅仅支持在线互动，还支持许多工具：悬停提示、选择区缩放、鼠标滚轮缩放、平移，也支持绘图存盘。

在 Bokeh 的展览区中，可以看到很多演示例子。

10.6.13　Mayavi

Mayavi 是 Enthought 开发的产品，其与 Autodesk 旗下的 Maya 无关，但它们都与 3D 建模有关。Mayavi 分为两个版本：Mayavi1 和 Mayavi2。

Mayavi2 是一套易用的 3D 数据交互可视化或者 3D 绘图软件包。它包括了以下特性：

- 可选的带有对话框的用户界面，用于在可视化图表中与数据和对象进行交互。

- Python 脚本接口，用于将 3D 可视化功能嵌入在应用中。
- 支持 VTK，但无须掌握 VTK 细节。
- 既可以作为单独程序，也可以作为扩展包嵌入在用户库和应用中。
- 以 2D/3D 形式对于标量、矢量和张量数据进行可视化。
- 通过客户源码、模块和数据过滤器进行扩展。
- 可读，如多种文件格式：VTK（旧格式与 XML）、PLOT3D 等。
- 可视化图表可以保存为多种图像格式。
- 通过 mlab 包提供三维图解函数，其功能类似于 matlab/matplotlib。

10.6.13.1 安装

和前面提到过的一些科学计算的 Python 包一样，虽然可以使用标准 pip 安装：

```
pip install mayavi
```

但是，Mayavi 会依赖于许多其他 Python 包，如 VTK、NumPy、Traits 等。所以，推荐使用 Python 发行版的现成安装包。

- Enthought Canopy；
- Python(x,y)；
- Anaconda。

10.6.13.2 算法代码生成徽标

要产生如图 10-14 所示的 Mayavi 徽标，可以运行以下代码。

mayavi_demo.py：

```
# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>
# Copyright (c) 2007, Enthought, Inc.
# License: BSD Style.


from numpy import sin, cos, mgrid, pi, sqrt
from mayavi import mlab

mlab.figure(fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))
u, v = mgrid[- 0.035:pi:0.01, - 0.035:pi:0.01]

X = 2 / 3. * (cos(u) * cos(2 * v) \
       + sqrt(2) * sin(u) * cos(v)) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
Y = 2 / 3. * (cos(u) * sin(2 * v) - \
       sqrt(2) * sin(u) * sin(v)) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
Z = -sqrt(2) * cos(u) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
S = sin(u)
```

```
mlab.mesh(X, Y, Z, scalars=S, colormap='YlGnBu', )

# Nice view from the front
mlab.view(.0, - 5.0, 4)
mlab.show()
```

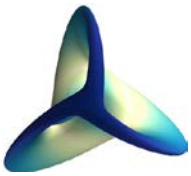


图 10-14　Mayavi 徽标，采用 Mayavi 代码产生

10.6.14　Vispy

Vispy 是针对交互式科学数据可视化而设计的快速、可扩展、易用的高性能 Python 软件包。它在底层通过 OpenGL 挖掘了 GPU 加速计算的能力。Vispy 的目标用户如下：

- OpenGL 用户，了解并愿意学习 OpenGL，并通过易学易用的 Python 来构建漂亮而高速的 2D/3D 可视化数据。掌握 OpenGL/GLSL 后，可以利用 vispy.gloo 来编写可视化插件。
- 数据科学家，但其可能不了解 OpenGL，希望寻找一套高层、高性能绘图工具箱。这些用户可以使用 vispy.plot 和 vispy.scene 接口来实现高层工作。
- 在图 10-15 中，Vispy 的 GPU 加速数据可视化演示了同时显示 320 组、每组 10K 数据点的音频波形。

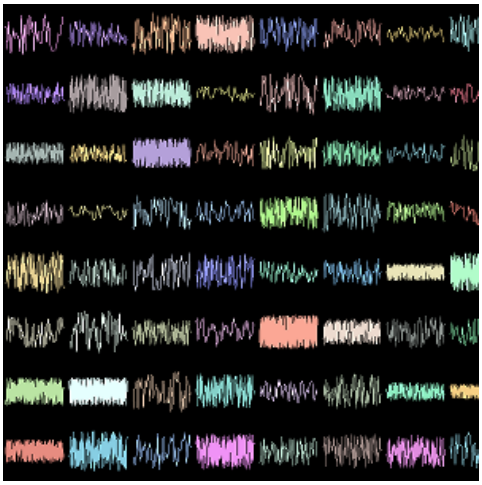


图 10-15　利用 GPU 加速绘图的 Vispy 演示例图

安装与测试

安装：

```
$ pip install --upgrade vispy
```

测试：

```
>>> import vispy
>>> vispy.test()
```

10.6.15 MoviePy

数据不仅仅是二维的。数据可视化配合时间轴可以形成动画，能够让人了解对象发展趋势，这是更有说服力的数据呈现技术。无论是 Mayavi 还是 Vispy 都只是产生 3D 动画，若要让人了解，则需要将其转换为适合传播的视频格式。

MoviePy 的开发目的适用于视频编辑，可用于基本操作，（如剪切、合并、插入标题），视频合成（如非线性编辑、视频处理），或者创建特效。它可以读/写基础视频格式（如 GIF 和 MP4）。MoviePy 是开源软件，其徽标如图 10-16 所示。

图 10-16　MoviePy 徽标

适用场景

具体场景如下：

- 有许多视频要批量处理，且需要以复杂形式采用程序进行合成；
- 在网络上提供视频处理自动化服务；
- 视频处理自动化任务，如插入字幕、标题、剪切视频、制作演职员表等；
- 添加视频特效；
- 为其他 Python 软件包（matplotlib、Mayavi、Gizeh、scikit-images）创建视频。

不适用场景

具体场景如下：

- 逐帧查看视频，实现人脸识别等高级任务，此类任务使用 OpenCV/SimpleCV 更合适；
- 格式转化，此类人物使用传统工具如 ffmpeg 更加快速。

优点和局限性

其优点如下：

- 简单易用，一行代码就可以完成基础操作；
- 使用灵活，完全掌握每一帧音频和视频，可使用 Python 创建自己的特效；
- 可移植性，基于常用软件如 NumPy、FFMEPG，可以在任意平台上运行。

其局限性如下：

- 目前，MoivePy 无法产生或处理视频流，无法产生 3D 特效，也不是为许多连续帧的视频处理而设计的软件。

movie_editor.py：

```
from moviepy.editor import *

# Load myHolidays.mp4 and select the subclip 00:00:50 - 00:00:60
clip = VideoFileClip("myHolidays.mp4").subclip(50,60)

# Generate a text clip (many options available ! )
txt_clip = TextClip("My Holidays 2013",fontsize=70,color='white')
txt_clip = txt_clip.set_pos('center').set_duration(10)

# Overlay the text clip above the first clip
final_clip = CompositeVideoClip([clip, txt_clip])

# write the result to a file in any format
final_clip.to_videofile("myHolidays_edited.avi",fps=25, codec='mpeg4')
```

MoviePy 的作者 Zulko 创造性地将其作为其他数据可视化软件包的后道动画渲染引擎，为数据可视化增加了动态视频输出效果。

10.6.16　其他新技术

随着数据分析平台和工具的快速迭代，数据可视化必将支持更多形式以实现更加直观的表达效果。可以展望的技术包括：

- 从多个维度动态“查看”数据分析结果；
- 配合 VR/AR 建模的数据可视化效果和设计；
- 基于 Leap Motion 和机器视觉的智能交互输入，以便更好地在 3D 环节中与数据可视化绘图进行交互。

10.7 本章小结

本章作为最后一章，主要介绍了数据融合、分析工具和平台以及数据可视化技术。希望本书的读者读完这些内容后，能够对于完整物联网各个环节的设计有个大概的了解，并有足够的兴趣去评估这些工具，集成在自己的工程项目中。

祝大家好运！

推荐书目与结束语

互联网、物联网近些年来呈现加速发展的态势，开发者需要不断地了解行业的进展，掌握最新的知识技能。其中，阅读相关书籍是非常必要的。

推荐书目

因为许多 Python 书籍都有入门介绍，因此关于入门书在此不再推荐。这里，笔者从自己手头的 Python 书籍中选择一批有特色的书籍介绍给大家。

通过阅读这些书籍，读者可以了解 Python 的进阶用法和标准库，以便在日常工程设计中采用高效、专业的编程方法，并可利用自动测试和文档完善自己的开发。在应用层面，读者可以掌握一些网络编程框架和运维，以及 GUI 编程等，并在科学计算、数据收集分析方面使用 Python 进行实际开发应用。市面上比较欠缺的是与硬件相关的图书，尤其是深嵌入 Python 的书籍较少，希望本书能够填补这一空白。

Python 语言进阶

- *The Python Standard Library by Examples*, by Doug Hellmann, Addison-Wesley
- *Expert Python Programming*, by Tarek Ziade, Packt
- *The Hacker's Guide to Python*, by Julien Danjou
- *Python High Performance Programming*, by Gabriele Lanro, Packt
- *The Practice of Computing Using Python*, by William F. Punch & Richard Enbody
- *Foundations of Agile Python Development*, by Jeff Younker, Apress
- *Python Testing Cookbook*, by Greg L. Turnquist, Packt
- *Effective Python, 59 Specific Ways to Write Better Python*, by Addison Wesley
- *Test-Driven Development with Python*, by Harry J.W. Percival, O'Reilly
- *PyMOTW Documentation*, by Doug Hellmann
- *Python Cookbook, David Beazley*, by Brian K Jones, O'Rellly

网站及运维

- *Twisted Network Programming Essential*, by Jessica McKellar & Abe Fettig, O'Reilly
- *Pro Python System Administration*, by Rytis Sileika, Apress
- *Flask Web Development*, by Miguel Grinberg, O'Reilly

硬件相关

- *Real World Instrumentation with Python*, by J.M. Hughes, O'Reilly
- *Raspberry Pi with Python*, by Simon Monk, McGraw Hill Education

应用相关

- *Pro Android Python with SL4A*, by Paul Ferrill, O'Reilly
- *Python Programming with Java Class Libraries*, by Addison Wesley
- *Hacking Secret Ciphers with Python*, by Al Sweigart
- *Jython, Python for the Java Platform*, by Apress
- *Rapid GUI Programming with Python and Qt*, by Mark Summerfield, Prentice Hall

分析统计

- *SciPy and NumPy*, by Eli Bressert, O'Reilly
- *Python for Data Analysis*, by Wes McKinney, O'Reilly
- *Python Scripting for Computational Science*, by Hans Petter Langtangen, University of Oslo
- *A Primer on Scientific Programming with Python*, by Hans Petter Langtangen, Springer-Verlag
- *Python for Finance*, by Yves Hilpisch, O'Reilly
- *Data Science from Scratch, with Python*, by Joel Grus, O'Reilly
- *Methods in Medical Informatics, Healthcare Programming in Perl, Python and Ruby*, by Jules J. Berman, CRC Press
- *MongoDB & Python*, by Niall O'Higgins, O'Reilly
- *Numberical Methods in Engineering with Python*, by Jaan Kiusalaas, Cambridge University Press

国内的 Python 书籍

- 《用 Python 做科学计算》，HYRY Studio 著
- 《Selenium2 Python 自动化测试实战》，虫师著

友情提示

物联网的发展在 2015 年开始出现加速演进的趋势。围绕着物联网的整个环节，发生了许多并购活动。在进行物联网设计时，应尽量回避由此而导致的货源和生态整合变化风险。

- 在半导体供应商中，NXP 和 Freescale 合并，Microchip 收购 Atmel，紫光也在打造中国半导体巨型企业。
- 在中间件供应商中，PTC 一口气收购了 Alexa/Coldlight 等一系列物联网和大数据供应商。
- 某些互联网企业大肆收购消费者场景 O2O 业务，并提供物联网各个环节的服务。
- 国外 IT 企业，如 Amazon、Microsoft、Google、IBM 继续强化云计算和 IoT 方面的投入。
- 跨界收购：Intel 收购了 Altera，Softbank 收购了 ARM。
- 其他行业也针对物联网提供垂直整合的方案。例如 Siemens、GE、Schineider 的工业物联网，CISCO、华为的各种物联网技术，以及美国 AT&T、日本 NTT、中国移动、中国联通和中国电信针对主要应用场景提供的物联网业务和平台等。

供应商和生态链平台

- 在选择半导体供应商设计节点设备和网关设备时，需要观察之前提到过的企业兼并重组，以及之后的产品线支持情况。
- 在采用各类云服务器（PaaS）供应商时，需要附加考察其资金情况，并在设计之初就要有 Plan B，做好云服务供应商随时倒闭或者变更服务计划的准备，这种准备需要在固件升级和服务器端设计中得到体现。
- 在提供产品和服务时，需要使用专利和专有技术保护自己，对供应商和客户保持警惕。

对传统企业的期望

本届政府在“两创”上寄予殷切期望，希望传统企业转型向“互联网+”发展。

在笔者以往的日常生产实践中，发现许多传统企业在选择物联网（连接性+云服务）方案时，内部观点繁杂，顾忌多，善变，纠结。这些企业往往纠结于各种连接性方案：2G、4G、Zigbee、Wi-Fi、PLC Modbus……由于不同供应商给出的技术方案和报价缺乏可比性，因此最终这些企业往往很难启动自己的物联网设计。

这也难怪这些企业，因为传统企业既缺乏微电子应用背景，又缺乏互联网或者通信行业基因。这些企业既要争夺行业制高点，又害怕其他行业对其进行颠覆。以传统企业的组织架构和

薪资水平，短期内要积累这方面的知识，招揽并留住人才，谈何容易。如果再裹挟着组织机构和外部协调关系，一个物联网项目往往会拖得非常长，以致最终流产。

企业主需要做的是：

- 定位清晰，包括中短期的市场定位和物联网的实际工程需求，这主要涉及节点数量、传输距离、数据流量和总量等特性。
- 根据定位和需求，选择合适的技术方案。
- 快速实施技术方案。如果是成熟方案，那么可以采用第三方方案，包括无线模块和 IoT 接入服务。
- 通过运营积累经验，深入了解自己的需求后，可以自建团队、自主研发物联网整体方案。

这一切的核心要素是人。如果一家组织结构要转型到物联网，甚至成为一家成功的物联网生态领军企业，首先需要从自身组织架构改变，变成物联网企业；否则在物联网时代，缺乏物联网的基因，就无法吸引人才，也无法形成自己的“护城河”优势。作为一家物联网企业，必须要有自己的核心技术团队。光靠外包是不行的。外包，核心业务也可以外包吗？！

物联网的特点是多中心化。但是某些拥有资金和网络技术优势的互联网企业很容易在局部市场如智能家居、可穿戴设备、酒店等应用领域形成垄断，压缩制造商的利润空间。但是在更多领域，如工业、农业、军事、教育领域、传统制造业的议价空间更大。如果选择合适的技术方案和平台，这些企业可以形成自己的行业中心平台。

所以，传统制造业要制定战略，选对连接技术，选对平台和系统服务供应商，不要过于封闭，这样才能够打造自己的行业物联网生态。虽然本书以 Python 为例，展开物联网的全链条开发，但是对于一家企业来说，如何吸引开发者和友商加入自家的行业生态，战略比战术要重要得多。

对自己的期待

物联网的环节长，需求也很旺盛。笔者自己的工程师思维很重，并一直在不断地学习各种技术，也做了不少项目。不过，笔者个人的时间和精力均十分有限，虽然笔者周围逐渐聚集了一些关注物联网发展的创业伙伴，但是对于如何切入物联网并构建一个持续发展的业务模式笔者一直没有考虑清楚：

- 依托开源设计开展设计与运维外包业务？
- 针对某个细分产品或者市场提供从 BOM 到云的交钥匙解决方案？
- 针对某些场景，提供前端数据整合？

- 针对物联网提供大数据服务？

这些问题在笔者埋头写稿和编写代码时没有细细考虑，希望本书付梓后，读者的反馈能够给自己带来更多的灵感。

教学相长

虽有嘉肴，弗食，不知其旨也；虽有至道，弗学，不知其善也。是故学然后知不足，教然后知困。知不足，然后能自反也；知困，然后能自强也。故曰：教学相长也。

节选自《礼记·学记》

在编写本书的时候，笔者不断发现新知识点，并补充到了自己的项目中。比如：进阶 Python 知识和各种 Python 技巧，Python 在 SPICE 和 HDL 方面的进展，MicroPython 在嵌入式 Linux 的使用，物联网网关的信号隧道，消息队列在嵌入式中的使用，大数据分析和可视化，等等。无论是自己熟悉的还是不熟悉的，都不断地刷新着笔者的视野。所以，笔者再次感谢电子工业出版社和编辑“永恒的侠少”让我有机会增长见识。

编写本书缘起笔者的心血来潮。坦白地说，独立编写如此厚度的一本书是对笔者能力的巨大挑战。笔者个人能力有限，欢迎广大读者指正错误。

结束语

本书除了介绍 Python 和物联网全栈开发，还有一个目的是推动开源。笔者将基于读者的反馈在以下环节中选定若干主题展开新的实验，推动开源工程。

博客内容

- 基于第 8 章展开并更新相关内容，围绕这一主题继续为物联网开发提供各类有用的工具软件。
- 基于 CPython、Jython、Cython 的 pystone 性能对比。
- 基于 CPython、Jython、Cython、libuv 的 Twisted 性能对比。
- 重现 Twisted/libuv 与 Golang/Node.js 的性能对比。
- InfluxDB、Cassandra、Hbase/OpenTSDB 等各类时序相关数据库技术实时查询性能对比。
- 基于 CUDA/OpenCL 的数据流实时分析包。
- 基于 Lambda 的数据分析实验。

硬件参考设计

- 硬件扩展模块：兼容 mikroBUS/USB/miniPCIe 的无线扩展板和配套模块，实现短距无线电和低功耗广域（如 LoRa）网卡的标准化和商品化。已经迭代了多种硬件与固件设计。
- Python MCU 主控板：基于 MicroPython/Viper/Zerynth 的共享 MCU，提供 mikroBUS/USB/miniPCIe 总线扩展接口的 MCU 应用板。
- Python/Jython 网关：通过 USB/UART/miniPCIe 将 IEEE 802.15.4/6LowPAN/LoRa 和各类工业总线模块接入 Linux SBC 设备，构建标准化网关的电气与外形接口。
- 定制服务：配合半导体供应商、设备制造商提供 PyMite/MicroPython/VIPER/Zerynth 的 PCBA，并增加各种无线连接模块选项。

固件/软件

- IoT 协议栈：WAN/WSN 相关协议 Python 软件包，用于各类 Python 物联网应用，主要依托 ARM mbed 平台发布。
- 基于虚拟机/Docker 支持 pymbed 在线编译。
- MicroPython 交叉编译：Linux 最小系统中的 MicroPython 支持。
- MicroPython 网关：基于 MicroPython 的物联网网关。
- MicroPython 手环：基于百度手环和 microbit 版本的 MicroPython 智能手环。
- Zerynth 网关：基于 VIPER/Zerynth 的物联网网关。
- Python 网关：基于 CPython/Jython 的物联网网关，测试并将 panStamp 等 Python 网关进行本地化和扩展，支持 LoRa/SubGHz 在细分行业的使用普及。
- 辅助工具集：针对嵌入式系统开发的 Python 辅助工具集。

设备云

- EPIC 发布：EPIC 的持续优化，尽快发布开源版本。
- EPIC 优化：测试 PyPy/pyuv 加速和 Golang 混合编程版本。
- EPIC 升级：提供配合主流 PaaS 的 IoT 前端提供 EPIC 版本。
- EPIC 实用化：EPIC 问卷式工程向导。
- EPIC 容器化：利用容器，实现 EPIC 快速部署。
- EPIC 平台化：配合半导体行业和分销商，针对特定行业提供交钥匙（Turn-key）工程，快速切入某些新型消费应用。

云端服务参考设计

- DaaS：Development as a Service。基于虚拟机/Docker 提供 PyMite/pymbed/MicroPython 编译平台。
- 参考应用服务器：基于 Flask+gevent，提供针对细分行业的 Python 应用云和 APP 的参

考实现。

- 参考移动 App：基于 Kivy、QPython 和 Web GUI，提供一些可立即使用的行业 APP。

希望大家持续关注和支持；更欢迎各方面的合作伙伴与笔者一起参与开源设计，拓展业务。

作者联络方式

Blog：http://blog.sina.com.cn/allankliu/

Mail：chip2cloud@163.com

GitHub：https://github.com/allankliu

Baidu：http://pan.baidu.com/s/1nvdHhmL 密码: hart